

```
// Collected Java Source Files
// Generated by AllJavaFilesCollector

// =====
// File: /net/elenamurat/light/LightProperties.java
// =====

package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

public class LightProperties {
    public final Vector3 direction;
    public final Color color;
    public final double intensity;

    public LightProperties(Vector3 direction, Color color, double intensity) {
        this.direction = direction;
        this.color = color;
        this.intensity = intensity;
    }

    // Factory method for ambient light
    public static LightProperties createAmbient(Color color, double
intensity) {
        return new LightProperties(new Vector3(0, 0, 0), color, intensity);
    }

    // Factory method for directional light
    public static LightProperties createDirectional(Vector3 direction, Color
color, double intensity) {
        return new LightProperties(direction.negate().normalize(), color,
intensity);
    }

    // Factory method for point light
```

```

public static LightProperties createPointLight(Vector3 lightPos, Point3
surfacePoint, Color color, double intensity) {
    Vector3 dir = lightPos.subtract(surfacePoint).normalize();
    return new LightProperties(dir, color, intensity);
}

// Null object pattern for safety
public static LightProperties nullProperties() {
    return new LightProperties(new Vector3(0, 1, 0), Color.BLACK, 0.0);
}

public static final LightProperties getLightProperties(Light light, Point3
point) {
    if (light == null) return nullProperties();

    if (light instanceof ElenaMuratAmbientLight) {
        return createAmbient(light.getColor(), light.getIntensity());
    }

    try {
        if (light instanceof MuratPointLight) {
            return createPointLight(
                light.getPosition().toVector(), point, light.getColor(),
                light.getAttenuatedIntensity(point)
            );
        }
        else if (light instanceof ElenaDirectionalLight) {
            return createDirectional(
                ((ElenaDirectionalLight)light).getDirection(), light.getColor(),
                light.getIntensity()
            );
        }
        else {
            return new LightProperties(
                new Vector3(0, 1, 0), light.getColor(), Math.min(light.getIntensity(),
                1.0)
            );
        }
    } catch (Exception e) {

```

```
        return nullProperties();
    }
}

}

// =====
// File: /net/elenamurat/light/FractalLight.java
// =====

package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.lovert.Scene;

public class FractalLight implements Light {
    private final Point3 position;
    private final Color baseColor;
    private final double baseIntensity;
    private final int octaves;
    private final double persistence;
    private final double frequency;
    private final double[] randoms;
    private final int[] permutations;

    public FractalLight(Point3 position, Color color, double intensity) {
        this(position, color, intensity, 4, 0.5, 0.1);
    }

    public FractalLight(Point3 position, Color color, double intensity,
        int octaves, double persistence, double frequency) {
        this.position = position;
        this.baseColor = color;
        this.baseIntensity = Math.max(0, intensity);
        this.octaves = Math.max(1, octaves);
        this.persistence = Math.max(0, Math.min(1, persistence));
    }
}
```

```
this.frequency = Math.max(0.001, frequency);
this.randoms = new double[256];
this.permutations = new int[512];

    initializeNoise();
}

private void initializeNoise() {
    for (int i = 0; i < 256; i++) {
        randoms[i] = Math.random() * 2 - 1;
        permutations[i] = i;
    }

    // Fisher-Yates shuffle
    for (int i = 255; i > 0; i--) {
        int index = (int)(Math.random() * (i + 1));
        int temp = permutations[i];
        permutations[i] = permutations[index];
        permutations[index] = temp;
    }

    // Duplicate for overflow
    System.arraycopy(permutations, 0, permutations, 256, 256);
}

@Override
public Point3 getPosition() {
    return position;
}

@Override
public Color getColor() {
    return baseColor;
}

@Override
public double getIntensity() {
    return baseIntensity;
}
```

```

@Override
public Vector3 getDirectionAt(Point3 point) {
    Vector3 direction = position.subtract(point);
    return direction.length() < Ray.EPSILON ? new Vector3(0,0,0) :
direction.normalize();
}

@Override
public double getAttenuatedIntensity(Point3 point) {
    double noise = fractalNoise(point.x, point.y, point.z);
    return baseIntensity * (0.3 + 0.7 * noise);
}

private double fractalNoise(double x, double y, double z) {
    double total = 0;
    double amplitude = 1.0;
    double maxAmplitude = 0;
    double freq = frequency;

    for (int i = 0; i < octaves; i++) {
        total += improvedNoise(x * freq, y * freq, z * freq) * amplitude;
        maxAmplitude += amplitude;
        amplitude *= persistence;
        freq *= 2.0;
    }

    return total / maxAmplitude;
}

private double improvedNoise(double x, double y, double z) {
    // Ken Perlin'in geliştirilmiş gürültü algoritması
    int xi = (int)Math.floor(x) & 255;
    int yi = (int)Math.floor(y) & 255;
    int zi = (int)Math.floor(z) & 255;

    double xf = x - Math.floor(x);
    double yf = y - Math.floor(y);
    double zf = z - Math.floor(z);
}

```

```

double u = fade(xf);
double v = fade(yf);
double w = fade(zf);

int aaa = permutations[permutations[permutations[xi] + yi] + zi];
int aba = permutations[permutations[permutations[xi] + yi + 1] + zi];
int aab = permutations[permutations[permutations[xi] + yi] + zi + 1];
int abb = permutations[permutations[permutations[xi] + yi + 1] + zi +
1];
    int baa = permutations[permutations[permutations[xi + 1] + yi] + zi];
    int bba = permutations[permutations[permutations[xi + 1] + yi + 1] +
zi];
    int bab = permutations[permutations[permutations[xi + 1] + yi] + zi +
1];
    int bbb = permutations[permutations[permutations[xi + 1] + yi + 1] + zi +
1];

double x1 = lerp(u, grad(aaa, xf, yf, zf), grad(baa, xf-1, yf, zf));
double x2 = lerp(u, grad(aba, xf, yf-1, zf), grad(bba, xf-1, yf-1, zf));
double y1 = lerp(v, x1, x2);

x1 = lerp(u, grad(aab, xf, yf, zf-1), grad(bab, xf-1, yf, zf-1));
x2 = lerp(u, grad(abb, xf, yf-1, zf-1), grad(bbb, xf-1, yf-1, zf-1));
double y2 = lerp(v, x1, x2);

return (lerp(w, y1, y2) + 1) / 2; // [-1,1] -> [0,1] aralığına normalize
}

private double fade(double t) {
    return t * t * t * (t * (t * 6 - 15) + 10);
}

private double lerp(double t, double a, double b) {
    return a + t * (b - a);
}

private double grad(int hash, double x, double y, double z) {
    int h = hash & 15;

```

```

        double u = h < 8 ? x : y;
        double v = h < 4 ? y : (h == 12 || h == 14 ? x : z);
        return ((h & 1) == 0 ? u : -u) + ((h & 2) == 0 ? v : -v);
    }

// Light interface diğer metodları
@Override
public Vector3 getDirectionTo(Point3 point) {
    Vector3 direction = point.subtract(position);
    return direction.length() < Ray.EPSILON ? new Vector3(0,0,0) :
direction.normalize();
}

@Override
public double getIntensityAt(Point3 point) {
    return getAttenuatedIntensity(point);
}

@Override
public boolean isVisibleFrom(Point3 point, Scene scene) {
    Vector3 lightDir = getDirectionTo(point);
    double distance = position.distance(point);
    Ray shadowRay = new Ray(
        point.add(lightDir.scale(Ray.EPSILON * 10)),
        lightDir
    );
    return !scene.intersects(shadowRay, distance - Ray.EPSILON);
}

}

// =====
// File: /net/elenamurat/light/SpotLight.java
// =====

package net.elena.murat.light;

import java.awt.Color;

```

```
import net.elena.murat.math.*;
import net.elena.murat.lovert.Scene;

public class SpotLight implements Light {
    private final Point3 position;
    private final Vector3 direction;
    private final Color color;
    private final double intensity;
    private final double cosInnerCone;
    private final double cosOuterCone;
    private final double constantAttenuation;
    private final double linearAttenuation;
    private final double quadraticAttenuation;

    public SpotLight(Point3 position, Vector3 direction, Color color,
        double intensity, double innerConeAngle, double outerConeAngle) {
        this(position, direction, color, intensity, innerConeAngle,
        outerConeAngle,
        1.0, 0.1, 0.01);
    }

    public SpotLight(Point3 position, Vector3 direction, Color color,
        double intensity, double innerConeAngle, double outerConeAngle,
        double constantAttenuation, double linearAttenuation, double
        quadraticAttenuation) {
        this.position = position;
        this.direction = direction.normalize();
        this.color = color;
        this.intensity = intensity;
        this.cosInnerCone = Math.cos(Math.toRadians(innerConeAngle/2));
        this.cosOuterCone = Math.cos(Math.toRadians(outerConeAngle/2));
        this.constantAttenuation = constantAttenuation;
        this.linearAttenuation = linearAttenuation;
        this.quadraticAttenuation = quadraticAttenuation;
    }

    @Override
    public Point3 getPosition() {
```

```
    return position;
}

@Override
public Color getColor() {
    return color;
}

@Override
public double getIntensity() {
    return intensity;
}

@Override
public Vector3 getDirectionAt(Point3 point) {
    return position.subtract(point).normalize();
}

@Override
public Vector3 getDirectionTo(Point3 point) {
    return point.subtract(position).normalize();
}

@Override
public double getAttenuatedIntensity(Point3 point) {
    double distance = position.distance(point);
    double attenuation = constantAttenuation +
        linearAttenuation * distance +
        quadraticAttenuation * distance * distance;
    double coneFactor = calculateConeFactor(point);
    return intensity * coneFactor / Math.max(attenuation, Ray.EPSILON);
}

@Override
public double getIntensityAt(Point3 point) {
    return getAttenuatedIntensity(point);
}

@Override
```

```
public boolean isVisibleFrom(Point3 point, Scene scene) {  
    Vector3 lightDir = getDirectionTo(point);  
    double distance = getDistanceTo(point);  
    Ray shadowRay = new Ray(  
        point.add(lightDir.scale(Ray.EPSILON * 10)),  
        lightDir  
    );  
    return !scene.intersects(shadowRay, distance - Ray.EPSILON);  
}  
  
public double getDistanceTo(Point3 point) {  
    return position.distance(point);  
}  
  
private double calculateConeFactor(Point3 point) {  
    Vector3 lightToPoint = getDirectionTo(point);  
    double dot = lightToPoint.dot(direction);  
  
    if (dot >= cosInnerCone) return 1.0;  
    if (dot <= cosOuterCone) return 0.0;  
  
    return (dot - cosOuterCone) / (cosInnerCone - cosOuterCone);  
}  
  
public double getInnerConeAngle() {  
    return Math.toDegrees(Math.acos(cosInnerCone)) * 2;  
}  
  
public double getOuterConeAngle() {  
    return Math.toDegrees(Math.acos(cosOuterCone)) * 2;  
}  
  
}  
  
// ======  
// File: /net/elenamurat/light/Light.java  
// ======
```

```
package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.lovert.Scene;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

public interface Light {

    Point3 getPosition();

    Color getColor();

    double getIntensity();

    Vector3 getDirectionAt(Point3 point);

    double getAttenuatedIntensity(Point3 point);

    double getIntensityAt(Point3 point);

    Vector3 getDirectionTo(Point3 point);

    boolean isVisibleFrom(Point3 point, Scene scene);

}
```

```
// =====
// File: /net/elena/murat/light/ElenaDirectionalLight.java
// =====
```

```
package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
```

```
import net.elena.murat.math.Ray;
import net.elena.murat.lovert.Scene;

public class ElenaDirectionalLight implements Light {
    private final Vector3 direction;
    private final Color color;
    private final double intensity;
    private static final double MIN_DIRECTION_LENGTH = 1e-6;

    public ElenaDirectionalLight(Vector3 direction, Color color, double
intensity) {
        if (direction == null || direction.length() <
MIN_DIRECTION_LENGTH) {
            throw new IllegalArgumentException("Direction cannot be null or
zero-length vector");
        }
        this.direction = direction.normalize();
        this.color = color != null ? color : Color.WHITE;
        this.intensity = Math.max(0, intensity);
    }

    @Override
    public Point3 getPosition() {
        return null; // Directional lights have no position
    }

    @Override
    public Color getColor() {
        return color;
    }

    @Override
    public double getIntensity() {
        return intensity;
    }

    @Override
    public Vector3 getDirectionAt(Point3 point) {
        return direction.negate();
```

```
}
```

```
@Override
```

```
public Vector3 getDirectionTo(Point3 point) {  
    return direction;  
}
```

```
@Override
```

```
public double getAttenuatedIntensity(Point3 point) {  
    return intensity; // No distance attenuation  
}
```

```
@Override
```

```
public double getIntensityAt(Point3 point) {  
    return intensity; // Uniform intensity everywhere  
}
```

```
@Override
```

```
public boolean isVisibleFrom(Point3 point, Scene scene) {  
    Ray shadowRay = new Ray(  
        point.add(direction.scale(Ray.EPSILON * 10)),  
        direction  
    );  
    return !scene.intersects(shadowRay, Double.POSITIVE_INFINITY);  
}
```

```
// Additional utility methods
```

```
public Vector3 getDirection() {  
    return direction;  
}
```

```
public ElenaDirectionalLight withDirection(Vector3 newDirection) {  
    return new ElenaDirectionalLight(newDirection, color, intensity);  
}
```

```
public ElenaDirectionalLight withColor(Color newColor) {  
    return new ElenaDirectionalLight(direction, newColor, intensity);  
}
```

```
public ElenaDirectionalLight withIntensity(double newIntensity) {
    return new ElenaDirectionalLight(direction, color, newIntensity);
}

public static ElenaDirectionalLight createDefault() {
    return new ElenaDirectionalLight(
        new Vector3(-1, -1, -1).normalize(),
        new Color(255, 255, 230),
        0.8
    );
}

@Override
public String toString() {
    return String.format(
        "DirectionalLight[direction=%s, color=%s, intensity=%.2f]",
        direction, color, intensity
    );
}

}

/**
// light like sun (from above)
ElenaDirectionalLight sunLight = new ElenaDirectionalLight(
    new Vector3(0, -1, 0.2).normalize(),
    new Color(255, 240, 220), // hot white
    1.2
);

// Light morning (color more hot)
ElenaDirectionalLight morningLight = sunLight
    .withColor(new Color(255, 220, 180))
    .withIntensity(0.8);
 */

// =====
// File: /net/elena/murat/light/PulsatingPointLight.java
```

```
// =====
```

```
package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Ray;
import net.elena.murat.lovert.Scene;

public class PulsatingPointLight implements Light {
    private final Point3 initialPosition;
    private final Color baseColor;
    private final double baseIntensity;
    private final double pulsationSpeed;
    private final double movementSpeed;
    private final double movementAmplitude;
    private final double constantAttenuation;
    private final double linearAttenuation;
    private final double quadraticAttenuation;

    private double currentTime;

    public PulsatingPointLight(Point3 initialPosition, Color baseColor,
double baseIntensity,
        double pulsationSpeed, double movementSpeed, double
movementAmplitude) {
        this(initialPosition, baseColor, baseIntensity, pulsationSpeed,
movementSpeed, movementAmplitude,
        1.0, 0.1, 0.01);
    }

    public PulsatingPointLight(Point3 initialPosition, Color baseColor,
double baseIntensity,
        double pulsationSpeed, double movementSpeed, double
movementAmplitude,
        double constantAttenuation, double linearAttenuation, double
quadraticAttenuation) {
```

```

this.initialPosition = initialPosition;
this.baseColor = baseColor;
this.baseIntensity = Math.max(0, baseIntensity);
this.pulsationSpeed = Math.max(0, pulsationSpeed);
this.movementSpeed = Math.max(0, movementSpeed);
this.movementAmplitude = Math.max(0, movementAmplitude);
this.constantAttenuation = Math.max(0, constantAttenuation);
this.linearAttenuation = Math.max(0, linearAttenuation);
this.quadraticAttenuation = Math.max(0, quadraticAttenuation);
this.currentTime = 0;
}

public void update(double deltaTime) {
    this.currentTime += deltaTime;
}

@Override
public Point3 getPosition() {
    double offsetX = Math.sin(currentTime * movementSpeed) *
movementAmplitude;
    double offsetY = Math.cos(currentTime * movementSpeed * 0.7) *
movementAmplitude * 0.5;
    double offsetZ = Math.sin(currentTime * movementSpeed * 0.3) *
movementAmplitude * 0.3;
    return new Point3(
        initialPosition.x + offsetX,
        initialPosition.y + offsetY,
        initialPosition.z + offsetZ
    );
}

@Override
public Color getColor() {
    double pulsationFactor = 0.7 + 0.3 * Math.sin(currentTime *
pulsationSpeed);
    return new Color(
        clampColor(baseColor.getRed() * pulsationFactor),
        clampColor(baseColor.getGreen() * pulsationFactor),
        clampColor(baseColor.getBlue() * pulsationFactor)
    );
}

```

```

    );
}

@Override
public double getIntensity() {
    return baseIntensity * (0.8 + 0.2 * Math.sin(currentTime *
pulsationSpeed * 1.3));
}

@Override
public Vector3 getDirectionAt(Point3 point) {
    return getPosition().subtract(point).normalize();
}

@Override
public Vector3 getDirectionTo(Point3 point) {
    return point.subtract(getPosition()).normalize();
}

@Override
public double getAttenuatedIntensity(Point3 point) {
    double distance = getDistanceTo(point);
    double attenuation = calculateAttenuation(distance);
    double pulsationFactor = 0.5 + 0.5 * Math.sin(currentTime *
pulsationSpeed * 1.7);
    return getIntensity() * pulsationFactor / Math.max(attenuation,
Ray.EPSILON);
}

@Override
public double getIntensityAt(Point3 point) {
    return getAttenuatedIntensity(point);
}

@Override
public boolean isVisibleFrom(Point3 point, Scene scene) {
    Vector3 lightDir = getDirectionTo(point);
    double distance = getDistanceTo(point);
    Ray shadowRay = new Ray(

```

```
        point.add(lightDir.scale(Ray.EPSILON * 10)),
        lightDir
    );
    return !scene.intersects(shadowRay, distance - Ray.EPSILON);
}

public double getDistanceTo(Point3 point) {
    return getPosition().distance(point);
}

private double calculateAttenuation(double distance) {
    return constantAttenuation +
        linearAttenuation * distance +
        quadraticAttenuation * distance * distance;
}

private int clampColor(double value) {
    return (int) Math.max(0, Math.min(255, value));
}

/**
// Titrek ates efekti
PulsatingPointLight fireLight = new PulsatingPointLight(
new Point3(0, 2, 0),
new Color(255, 100, 50), // Turuncu
1.5, // Temel yoğunluk
2.0, // Hızlı titreme
0.5, // Orta hızda hareket
0.3 // Hafif salınım
);

// Her frame'de guncelle
fireLight.update(deltaTime);

// For all lights in scene:
// pointLight.getIntensityAt(hitPoint)...
*/
```

```
// =====
// File: /net/elenamurat/light/ElenaMuratAmbientLight.java
// =====

package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Ray;
import net.elena.murat.lovert.Scene;

public class ElenaMuratAmbientLight implements Light {
    private final Color color;
    private final double intensity;
    private static final Vector3 ZERO_VECTOR = new Vector3(0, 0, 0);

    public ElenaMuratAmbientLight(Color color, double intensity) {
        this.color = color != null ? color : new Color(200, 220, 255);
        this.intensity = Math.max(0, Math.min(1, intensity));
    }

    @Override
    public Point3 getPosition() {
        return null;
    }

    @Override
    public Color getColor() {
        return color;
    }

    @Override
    public double getIntensity() {
        return intensity;
    }
}
```

```
@Override
public Vector3 getDirectionAt(Point3 point) {
    return ZERO_VECTOR;
}

@Override
public Vector3 getDirectionTo(Point3 point) {
    return ZERO_VECTOR;
}

@Override
public double getAttenuatedIntensity(Point3 point) {
    return intensity;
}

@Override
public double getIntensityAt(Point3 point) {
    return intensity;
}

@Override
public boolean isVisibleFrom(Point3 point, Scene scene) {
    return true; // Ambient light is always visible
}

// Utility methods
public ElenaMuratAmbientLight withColor(Color newColor) {
    return new ElenaMuratAmbientLight(newColor, intensity);
}

public ElenaMuratAmbientLight withIntensity(double newIntensity) {
    return new ElenaMuratAmbientLight(color, newIntensity);
}

public static ElenaMuratAmbientLight createDefault() {
    return new ElenaMuratAmbientLight(new Color(200, 220, 255), 0.15);
}

@Override
```

```
public String toString() {
    return String.format("AmbientLight[color=%s, intensity=%.2f]", color,
intensity);
}

}

// =====
// File: /net/elenamurat/light/BlackHoleLight.java
// =====

package net.elena.murat.light;

import java.awt.Color;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Ray;
import net.elena.murat.lovert.Scene;

public class BlackHoleLight implements Light {
    private final Point3 singularity;
    private final double eventHorizonRadius;
    private final Color accretionColor;
    private final double baseIntensity;
    private static final double GRAVITATIONAL_WARP_FACTOR = 2.0;

    public BlackHoleLight(Point3 singularity, double radius, Color color) {
        this(singularity, radius, color, 1.5);
    }

    public BlackHoleLight(Point3 singularity, double radius, Color color,
double intensity) {
        if (singularity == null) {
            throw new IllegalArgumentException("Singularity point cannot be
null");
        }
        this.singularity = singularity;
```

```
    this.eventHorizonRadius = Math.max(0.1, radius);
    this.accretionColor = color != null ? color : new Color(200, 150, 255);
    this.baseIntensity = Math.max(0, intensity);
}
```

```
@Override
public Point3 getPosition() {
    return singularity;
}
```

```
@Override
public Color getColor() {
    return accretionColor;
}
```

```
@Override
public double getIntensity() {
    return baseIntensity;
}
```

```
@Override
public Vector3 getDirectionAt(Point3 point) {
    Vector3 dir = singularity.subtract(point);
    double dist = dir.length();
    if (dist < Ray.EPSILON) {
        return new Vector3(0, 0, 0);
    }
    double warpFactor = GRAVITATIONAL_WARP_FACTOR / (1.0 -
Math.exp(-dist/eventHorizonRadius));
    return dir.normalize().multiply(warpFactor);
}
```

```
@Override
public Vector3 getDirectionTo(Point3 point) {
    Vector3 dir = point.subtract(singularity);
    double dist = dir.length();
    if (dist < Ray.EPSILON) {
        return new Vector3(0, 0, 0);
    }
}
```

```

        return dir.normalize();
    }

@Override
public double getAttenuatedIntensity(Point3 point) {
    double distance = singularity.distance(point);
    if (distance < eventHorizonRadius) {
        return 0.0;
    }
    return baseIntensity * (1.0 - eventHorizonRadius/distance);
}

@Override
public double getIntensityAt(Point3 point) {
    return getAttenuatedIntensity(point);
}

@Override
public boolean isVisibleFrom(Point3 point, Scene scene) {
    if (isPointBeyondEventHorizon(point)) {
        return false;
    }
    Vector3 lightDir = getDirectionTo(point);
    Ray shadowRay = new Ray(
        point.add(lightDir.scale(Ray.EPSILON * 10)),
        lightDir
    );
    return !scene.intersects(shadowRay, Double.POSITIVE_INFINITY);
}

public double getEventHorizonRadius() {
    return eventHorizonRadius;
}

public boolean isPointBeyondEventHorizon(Point3 point) {
    return singularity.distance(point) < eventHorizonRadius;
}

// Utility methods

```

```
public BlackHoleLight withSingularity(Point3 newSingularity) {
    return new BlackHoleLight(newSingularity, eventHorizonRadius,
accretionColor, baseIntensity);
}

public BlackHoleLight withRadius(double newRadius) {
    return new BlackHoleLight(singularity, newRadius, accretionColor,
baseIntensity);
}

public BlackHoleLight withColor(Color newColor) {
    return new BlackHoleLight(singularity, eventHorizonRadius, newColor,
baseIntensity);
}

public BlackHoleLight withIntensity(double newIntensity) {
    return new BlackHoleLight(singularity, eventHorizonRadius,
accretionColor, newIntensity);
}

@Override
public String toString() {
    return String.format(
        "BlackHoleLight[singularity=%s, radius=%.2f, color=%s, intensity=\
%.2f]",
        singularity, eventHorizonRadius, accretionColor, baseIntensity
    );
}

}

// =====
// File: /net/lena/murat/light/MuratPointLight.java
// =====
```

```
package net.elena.murat.light;
```

```
import java.awt.Color;
```

```
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Ray;
import net.elena.murat.lovert.Scene;

public class MuratPointLight implements Light {
    private final Point3 position;
    private final Color color;
    private final double intensity;
    private final double constantAttenuation;
    private final double linearAttenuation;
    private final double quadraticAttenuation;

    public MuratPointLight(Point3 position, Color color, double intensity) {
        this(position, color, intensity, 1.0, 0.1, 0.01);
    }

    public MuratPointLight(Point3 position, Color color, double intensity,
                           double constantAttenuation, double linearAttenuation, double
                           quadraticAttenuation) {
        this.position = position;
        this.color = color;
        this.intensity = Math.max(0, intensity);
        this.constantAttenuation = Math.max(0, constantAttenuation);
        this.linearAttenuation = Math.max(0, linearAttenuation);
        this.quadraticAttenuation = Math.max(0, quadraticAttenuation);
    }

    @Override
    public Point3 getPosition() {
        return position;
    }

    @Override
    public Color getColor() {
        return color;
    }
}
```

```
@Override
public double getIntensity() {
    return intensity;
}

@Override
public Vector3 getDirectionAt(Point3 point) {
    Vector3 direction = position.subtract(point);
    return direction.length() < Ray.EPSILON ? new Vector3(0,0,0) :
direction.normalize();
}

@Override
public Vector3 getDirectionTo(Point3 point) {
    Vector3 direction = point.subtract(position);
    return direction.length() < Ray.EPSILON ? new Vector3(0,0,0) :
direction.normalize();
}

@Override
public double getAttenuatedIntensity(Point3 point) {
    double distance = position.distance(point);
    double attenuation = constantAttenuation +
linearAttenuation * distance +
quadraticAttenuation * distance * distance;
    return intensity / Math.max(attenuation, Ray.EPSILON);
}

@Override
public double getIntensityAt(Point3 point) {
    return getAttenuatedIntensity(point);
}

@Override
public boolean isVisibleFrom(Point3 point, Scene scene) {
    Vector3 lightDir = getDirectionTo(point);
    double distance = getDistanceTo(point);
    Ray shadowRay = new Ray(
        point.add(lightDir.scale(Ray.EPSILON * 10)),
```

```

        lightDir
    );
    return !scene.intersects(shadowRay, distance - Ray.EPSILON);
}

public double getDistanceTo(Point3 point) {
    return position.distance(point);
}
}

/**
MuratPointLight light = new MuratPointLight(
new Point3(2, 5, -3),
Color.WHITE,
1.5
);

MuratPointLight longRangeLight = new MuratPointLight(
new Point3(0, 10, 0),
new Color(255, 220, 180),
2.0,
1.0, // c1
0.02, // c2
0.001 // c3
);
*/

```

```

// =====
// File: /net/elenamurat/light/BioluminescentLight.java
// =====

```

```

package net.elena.murat.light;

import java.awt.Color;
import java.util.Collections;
import java.util.List;

import net.elena.murat.math.Point3;

```

```

import net.elena.murat.math.Vector3;
import net.elena.murat.math.Ray;
import net.elena.murat.lovert.Scene;

public class BioluminescentLight implements Light {
    private final List<Point3> organismPositions;
    private final Color baseColor;
    private final double pulseSpeed;
    private final double baseIntensity;
    private final double attenuationFactor;
    private double currentTime;
    private static final double MIN_PULSE_INTENSITY = 0.7;
    private static final double PULSE_AMPLITUDE = 0.3;

    public BioluminescentLight(List<Point3> positions, Color color, double
pulseSpeed) {
        this(positions, color, pulseSpeed, 0.8, 0.2);
    }

    public BioluminescentLight(List<Point3> positions, Color color,
double pulseSpeed, double baseIntensity, double attenuationFactor) {
        if (positions == null || positions.isEmpty()) {
            throw new IllegalArgumentException("Organism positions cannot be
null or empty");
        }
        this.organismPositions = Collections.unmodifiableList(positions);
        this.baseColor = color != null ? color : new Color(100, 255, 150);
        this.pulseSpeed = Math.max(0, pulseSpeed);
        this.baseIntensity = Math.max(0, baseIntensity);
        this.attenuationFactor = Math.max(0, attenuationFactor);
    }

    public void update(double deltaTime) {
        this.currentTime += deltaTime;
    }

    @Override
    public Point3 getPosition() {
        return organismPositions.get(0); // Reference position for the light
    }
}

```

```
}
```

```
@Override  
public Color getColor() {  
    double pulseFactor = MIN_PULSE_INTENSITY +  
    PULSE_AMPLITUDE * Math.sin(currentTime * pulseSpeed);  
    return new Color(  
        clampColor(baseColor.getRed() * pulseFactor),  
        clampColor(baseColor.getGreen() * pulseFactor),  
        clampColor(baseColor.getBlue() * pulseFactor)  
    );  
}
```

```
@Override  
public double getIntensity() {  
    return baseIntensity * (MIN_PULSE_INTENSITY +  
    PULSE_AMPLITUDE * Math.sin(currentTime * pulseSpeed * 1.2));  
}
```

```
@Override  
public Vector3 getDirectionAt(Point3 point) {  
    Point3 closest = findClosestPosition(point);  
    return closest.subtract(point).normalize();  
}
```

```
@Override  
public Vector3 getDirectionTo(Point3 point) {  
    Point3 closest = findClosestPosition(point);  
    return point.subtract(closest).normalize();  
}
```

```
@Override  
public double getAttenuatedIntensity(Point3 point) {  
    double minDistance = findClosestPosition(point).distance(point);  
    return getIntensity() / (1.0 + attenuationFactor * minDistance);  
}
```

```
@Override  
public double getIntensityAt(Point3 point) {
```

```

        return getAttenuatedIntensity(point);
    }

@Override
public boolean isVisibleFrom(Point3 point, Scene scene) {
    Point3 closest = findClosestPosition(point);
    Vector3 lightDir = point.subtract(closest).normalize();
    double distance = closest.distance(point);

    Ray shadowRay = new Ray(
        point.add(lightDir.scale(Ray.EPSILON * 10)),
        lightDir
    );
    return !scene.intersects(shadowRay, distance - Ray.EPSILON);
}

public double getClosestDistance(Point3 point) {
    return findClosestPosition(point).distance(point);
}

private Point3 findClosestPosition(Point3 point) {
    return organismPositions.stream()
        .min((p1, p2) -> Double.compare(p1.distance(point),
        p2.distance(point)))
        .orElse(organismPositions.get(0));
}

private int clampColor(double value) {
    return (int) Math.max(0, Math.min(255, value));
}

// Utility methods
public BioluminescentLight withPositions(List<Point3> newPositions) {
    return new BioluminescentLight(newPositions, baseColor, pulseSpeed,
        baseIntensity, attenuationFactor);
}

public BioluminescentLight withColor(Color newColor) {
    return new BioluminescentLight(organismPositions, newColor,

```

```
        pulseSpeed, baseIntensity, attenuationFactor);
    }

    public BioluminescentLight withPulseSpeed(double newSpeed) {
        return new BioluminescentLight(organismPositions, baseColor,
newSpeed, baseIntensity, attenuationFactor);
    }

    public static BioluminescentLight createDefault() {
        return new BioluminescentLight(
            Collections.singletonList(new Point3(0, 0, 0)), // Tek elemanlı liste
            new Color(100, 255, 150),
            1.5
        );
    }

    @Override
    public String toString() {
        return String.format(
            "BioluminescentLight[positions=%d, color=%s, pulseSpeed=%.2f]",
            organismPositions.size(), baseColor, pulseSpeed
        );
    }

}

// =====
// File: /net/elenamurat/util/ColorUtil.java
// =====

package net.elena.murat.util;

import java.awt.Color;

/**
 * Utility class for RGB color operations.
 * All methods work exclusively with RGB components (alpha channel is
 * ignored).

```

```

* Results are always fully opaque (alpha=255).
*/
public final class ColorUtil {

    // Color constants (all fully opaque)
    public static final Color BLACK = new Color(0, 0, 0);
    public static final Color WHITE = new Color(255, 255, 255);
    public static final Color RED = new Color(255, 0, 0);
    public static final Color GREEN = new Color(0, 255, 0);
    public static final Color BLUE = new Color(0, 0, 255);

    /**
     * Linear interpolation between two RGB colors (t=0: c1, t=1: c2)
     * Only RGB components are interpolated, result is fully opaque
     */
    public static Color lerp(Color c1, Color c2, float t) {
        t = clamp(t, 0.0f, 1.0f);
        return new Color(
            c1.getRed() + (int)((c2.getRed() - c1.getRed()) * t),
            c1.getGreen() + (int)((c2.getGreen() - c1.getGreen()) * t),
            c1.getBlue() + (int)((c2.getBlue() - c1.getBlue()) * t)
        );
    }

    /**
     * Component-wise multiplication of two RGB colors
     * Result is always fully opaque (alpha=255)
     */
    public static Color multiply(Color c1, Color c2) {
        return new Color(
            (c1.getRed() * c2.getRed()) / 255,
            (c1.getGreen() * c2.getGreen()) / 255,
            (c1.getBlue() * c2.getBlue()) / 255
        );
    }

    /**
     * Multiplies RGB color by a scalar factor
     * Result is always fully opaque (alpha=255)
     */
}

```

```

*/
public static Color multiply(Color c, float scalar) {
    scalar = Math.max(0.0f, scalar);
    return new Color(
        clamp((int)(c.getRed() * scalar)),
        clamp((int)(c.getGreen() * scalar)),
        clamp((int)(c.getBlue() * scalar))
    );
}

/***
 * Reinhard Tone Mapping Operator
 */
public static float reinhardToneMap(float color) {
    return color / (1.0f + color);
}

public static Color gammaCorrect(Color color, float gamma) {
    float invGamma = 1.0f / gamma;
    float r = (float) Math.pow(color.getRed() / 255.0, invGamma);
    float g = (float) Math.pow(color.getGreen() / 255.0, invGamma);
    float b = (float) Math.pow(color.getBlue() / 255.0, invGamma);
    float a = color.getAlpha() / 255.0f;

    // Garanti için clamp (NaN veya aşırı değerlere karşı)
    r = Math.max(0.0f, Math.min(1.0f, r));
    g = Math.max(0.0f, Math.min(1.0f, g));
    b = Math.max(0.0f, Math.min(1.0f, b));
    a = Math.max(0.0f, Math.min(1.0f, a));

    return new Color(r, g, b, a);
}

private static float linearToSrgb(float linear) {
    if (linear <= 0.0031308f) {
        return linear * 12.92f;
    } else {
        return (float) (1.055f * Math.pow(linear, 1.0/2.4) - 0.055f);
    }
}

```

```

}

/***
 * sRGB to linear conversion for a single channel
 */
public static float srgbToLinear(float srgb) {
    if (srgb <= 0.04045f) {
        return srgb / 12.92f;
    } else {
        return (float) Math.pow((srgb + 0.055f) / 1.055f, 2.4f);
    }
}

public static Color sRGBToLinear(Color srgbColor, float gamma) {
    if (gamma == 1f) return srgbColor;

    float r = srgbColor.getRed() / 255.0f;
    float g = srgbColor.getGreen() / 255.0f;
    float b = srgbColor.getBlue() / 255.0f;
    float a = srgbColor.getAlpha() / 255.0f;

    r = clamp(r, 0.0f, 1.0f);
    g = clamp(g, 0.0f, 1.0f);
    b = clamp(b, 0.0f, 1.0f);

    r = (r <= 0.04045f) ? (r / 12.92f) : (float) Math.pow((r + 0.055f) / 1.055f,
    gamma);
    g = (g <= 0.04045f) ? (g / 12.92f) : (float) Math.pow((g + 0.055f) /
    1.055f, gamma);
    b = (b <= 0.04045f) ? (b / 12.92f) : (float) Math.pow((b + 0.055f) /
    1.055f, gamma);

    return new Color(r, g, b, a);
}

public static Color sRGBToLinearExtra(Color base, float gamma) {
    if (gamma == 1f) return base;

    if (gamma <= 0) gamma = 2.2f;
}

```

```

float r = base.getRed() / 255.0f;
float g = base.getGreen() / 255.0f;
float b = base.getBlue() / 255.0f;
float a = base.getAlpha() / 255.0f;

r = (float) Math.pow(r, 1.0f / gamma);
g = (float) Math.pow(g, 1.0f / gamma);
b = (float) Math.pow(b, 1.0f / gamma);

int red = (int) (Math.min(Math.max(r * 255, 0), 255));
int green = (int) (Math.min(Math.max(g * 255, 0), 255));
int blue = (int) (Math.min(Math.max(b * 255, 0), 255));
int alpha = (int) (a * 255);

return new Color(red, green, blue, alpha);
}

/***
 * Apply exposure and tone mapping to linear color
 */
public static Color applyExposureAndToneMapping(Color linearColor,
float exposure) {
    float r = clamp(linearColor.getRed(), 0.0f, 1.0f);
    float g = clamp(linearColor.getGreen(), 0.0f, 1.0f);
    float b = clamp(linearColor.getBlue(), 0.0f, 1.0f);
    float a = clamp(linearColor.getAlpha(), 0.0f, 1.0f);

    // Exposure adjustment
    r *= exposure;
    g *= exposure;
    b *= exposure;

    // ACES filmic tone mapping
    r = clamp(acesToneMap(r), 0.0f, 1.0f);
    g = clamp(acesToneMap(g), 0.0f, 1.0f);
    b = clamp(acesToneMap(b), 0.0f, 1.0f);

    return new Color(r, g, b, a);
}

```

```

}

/***
 * Apply tone mapping to a linear color
 */
public static Color applyToneMapping(Color linearColor, float exposure)
{
    float r = clamp(linearColor.getRed() / 255.0f, 0.0f, 1.0f);
    float g = clamp(linearColor.getGreen() / 255.0f, 0.0f, 1.0f);
    float b = clamp(linearColor.getBlue() / 255.0f, 0.0f, 1.0f);
    float a = clamp(linearColor.getAlpha() / 255.0f, 0.0f, 1.0f);

    // Exposure adjustment
    r *= exposure;
    g *= exposure;
    b *= exposure;

    // Reinhard tone mapping
    r = clamp(reinhardToneMap(r), 0.0f, 1.0f);
    g = clamp(reinhardToneMap(g), 0.0f, 1.0f);
    b = clamp(reinhardToneMap(b), 0.0f, 1.0f);

    return new Color(r, g, b, a);
}

public static Color linearToSRGB(Color linearColor) {
    float r = clamp(linearColor.getRed(), 0.0f, 1.0f);
    float g = clamp(linearColor.getGreen(), 0.0f, 1.0f);
    float b = clamp(linearColor.getBlue(), 0.0f, 1.0f);
    float a = clamp(linearColor.getAlpha(), 0.0f, 1.0f);

    // Linear to sRGB conversion
    r = clamp(linearToSrgb(r), 0.0f, 1.0f);
    g = clamp(linearToSrgb(g), 0.0f, 1.0f);
    b = clamp(linearToSrgb(b), 0.0f, 1.0f);

    return new Color(r, g, b, a);
}

```

```

private static float acesToneMap(float x) {
    float a = 2.51f;
    float b = 0.03f;
    float c = 2.43f;
    float d = 0.59f;
    float e = 0.14f;
    return Math.max(0.0f, Math.min(1.0f, (x * (a * x + b)) / (x * (c * x + d)
+ e)));
}

public static Color enhanceColorSaturation(Color color, float
saturationFactor) {
    float[] hsb = Color.RGBtoHSB(color.getRed(), color.getGreen(),
color.getBlue(), null);
    hsb[1] = Math.min(1.0f, hsb[1] * saturationFactor); // Increase
saturation
    return new Color(Color.HSBtoRGB(hsb[0], hsb[1], hsb[2]));
}

public static Color applyShadowColor(Color original, Color
shadowColor) {
    float alpha = shadowColor.getAlpha() / 255.0f;
    return new Color(
        (int)(shadowColor.getRed() * alpha + original.getRed() * (1 - alpha)),
        (int)(shadowColor.getGreen() * alpha + original.getGreen() * (1 -
alpha)),
        (int)(shadowColor.getBlue() * alpha + original.getBlue() * (1 -
alpha))
    );
}

public static Color enhanceBrightnessAndContrast(Color color, float
brightnessFactor, float contrastFactor) {
    int r = color.getRed();
    int g = color.getGreen();
    int b = color.getBlue();

    // Apply brightness
    r = Math.min(255, (int)(r * brightnessFactor));

```

```

g = Math.min(255, (int)(g * brightnessFactor));
b = Math.min(255, (int)(b * brightnessFactor));

// Apply contrast
float contrast = (contrastFactor - 1.0f) / 2.0f;
r = (int)((r - 128) * contrastFactor + 128 + contrast * 255);
g = (int)((g - 128) * contrastFactor + 128 + contrast * 255);
b = (int)((b - 128) * contrastFactor + 128 + contrast * 255);

// Clamp values
r = Math.max(0, Math.min(255, r));
g = Math.max(0, Math.min(255, g));
b = Math.max(0, Math.min(255, b));

return new Color(r, g, b, color.getAlpha());
}

/***
 * Scales RGB components by a factor (0.0-1.0)
 * Result is always fully opaque (alpha=255)
 */
public static Color multiplyColor(Color color, double factor) {
    factor = Math.max(0, Math.min(1, factor));
    return new Color(
        (int)(color.getRed() * factor),
        (int)(color.getGreen() * factor),
        (int)(color.getBlue() * factor)
    );
}

public static Color multiplyColorFloat(Color color, float factor) {
    factor = Math.max(0f, Math.min(1f, factor));
    return new Color(
        (int)(color.getRed() * factor),
        (int)(color.getGreen() * factor),
        (int)(color.getBlue() * factor)
    );
}

```

```
public static Color multiplyColors(Color color1, Color color2) {  
    float r = color1.getRed() / 255.0f * color2.getRed() / 255.0f;  
    float g = color1.getGreen() / 255.0f * color2.getGreen() / 255.0f;  
    float b = color1.getBlue() / 255.0f * color2.getBlue() / 255.0f;  
  
    return new Color(r, g, b);  
}  
  
/**  
 * Multiplies two colors with a scaling factor  
 * Result is always fully opaque (alpha=255)  
 */  
public static Color multiplyColors(Color base, Color light, double factor)  
{  
    int r = (int) Math.min(255, Math.max(0, base.getRed() * light.getRed() /  
255.0 * factor));  
    int g = (int) Math.min(255, Math.max(0, base.getGreen() *  
light.getGreen() / 255.0 * factor));  
    int b = (int) Math.min(255, Math.max(0, base.getBlue() *  
light.getBlue() / 255.0 * factor));  
  
    return new Color(r, g, b);  
}  
  
/**  
 * Creates a Color object from double values with robust validation  
 * Result is always fully opaque (alpha=255)  
 */  
public static Color createColor(double r, double g, double b) {  
    if (Double.isNaN(r) || Double.isNaN(g) || Double.isNaN(b)) {  
        return BLACK;  
    }  
  
    return new Color(  
        clamp((int)r),  
        clamp((int)g),  
        clamp((int)b)  
    );  
}
```

```
/**  
 * Clamps double value to [0, 255] range and rounds to nearest integer  
 */  
private static int clampAndRound(double value) {  
    if (value > Double.MAX_VALUE / 2) return 255;  
    if (value < -Double.MAX_VALUE / 2) return 0;  
  
    double clamped = Math.max(0.0, Math.min(255.0, value));  
    return (int) Math.round(clamped);  
}  
  
public static int clampColorValue(int value) {  
    if (value < 0) {  
        return 0;  
    }  
    if (value > 255) {  
        return 255;  
    }  
    return value;  
}  
  
public static float clampFloatColorValue(float value) {  
    if (value < 0f) {  
        return 0f;  
    }  
    if (value > 1f) {  
        return 1f;  
    }  
    return value;  
}  
  
public static Color clampColor(Color color) {  
    int r = Math.max(0, Math.min(255, color.getRed()));  
    int g = Math.max(0, Math.min(255, color.getGreen()));  
    int b = Math.max(0, Math.min(255, color.getBlue()));  
    return new Color(r, g, b);  
}
```

```

// Overload for float values
public static Color createColor(float r, float g, float b) {
    return createColor((double) r, (double) g, (double) b);
}

// Overload for int values
public static Color createColor(int r, int g, int b) {
    return new Color(
        Math.min(255, Math.max(0, r)),
        Math.min(255, Math.max(0, g)),
        Math.min(255, Math.max(0, b))
    );
}

// Interpolate between two colors
public static Color interpolateColor(Color c1, Color c2, double t) {
    return blendColors(c1, c2, t);
}

// Combine multiple colors (additive blending)
public static Color combineColors(Color... colors) {
    int r = 0, g = 0, b = 0;
    for (Color c : colors) {
        r = Math.min(255, r + c.getRed());
        g = Math.min(255, g + c.getGreen());
        b = Math.min(255, b + c.getBlue());
    }
    return new Color(r, g, b);
}

// Add noise/variation to a color
public static Color addColorVariation(Color color, double variation) {
    double noise = 0.9 + Math.sin(variation * 15.0) * 0.1;
    int r = (int)(color.getRed() * noise);
    int g = (int)(color.getGreen() * noise);
    int b = (int)(color.getBlue() * noise);
    return new Color(clamp(r), clamp(g), clamp(b));
}

```

```

// Darken a color by specified amount (0.0 - 1.0)
public static Color darkenColor(Color color, double amount) {
    amount = Math.max(0, Math.min(1, amount));
    int r = (int)(color.getRed() * (1 - amount));
    int g = (int)(color.getGreen() * (1 - amount));
    int b = (int)(color.getBlue() * (1 - amount));
    return new Color(clamp(r), clamp(g), clamp(b));
}

// Lighten a color by specified amount (0.0 - 1.0)
public static Color lightenColor(Color color, double amount) {
    amount = Math.max(0, Math.min(1, amount));
    int r = (int)(color.getRed() + (255 - color.getRed()) * amount);
    int g = (int)(color.getGreen() + (255 - color.getGreen()) * amount);
    int b = (int)(color.getBlue() + (255 - color.getBlue()) * amount);
    return new Color(clamp(r), clamp(g), clamp(b));
}

/***
 * Color addition (RGB only)
 * Result is always fully opaque (alpha=255)
 */
public static Color add(Color c1, Color c2) {
    return new Color(
        Math.min(255, c1.getRed() + c2.getRed()),
        Math.min(255, c1.getGreen() + c2.getGreen()),
        Math.min(255, c1.getBlue() + c2.getBlue())
    );
}

/***
 * Extracts float components [R,G,B] from AWT Color (0.0-1.0 range)
 */
public static float[] getFloatComponents(Color color) {
    float[] comp = new float[3];
    comp[0] = color.getRed() / 255.0f;
    comp[1] = color.getGreen() / 255.0f;
    comp[2] = color.getBlue() / 255.0f;
    return comp;
}

```

```

}

/***
 * Adds specular highlight effect to a color based on intensity
 * Result is always fully opaque (alpha=255)
 */
public static Color addSpecularHighlight(Color baseColor, double
intensity) {
    intensity = Math.max(0, Math.min(1, intensity));
    int r = (int)(baseColor.getRed() + (255 - baseColor.getRed()) *
intensity);
    int g = (int)(baseColor.getGreen() + (255 - baseColor.getGreen()) *
intensity);
    int b = (int)(baseColor.getBlue() + (255 - baseColor.getBlue()) *
intensity);
    return new Color(clamp(r), clamp(g), clamp(b));
}

/***
 * Adds specular highlight with custom highlight color
 * Result is always fully opaque (alpha=255)
 */
public static Color addSpecularHighlight(Color baseColor, Color
highlightColor, double intensity) {
    intensity = Math.max(0, Math.min(1, intensity));
    int r = (int)(baseColor.getRed() + (highlightColor.getRed() -
baseColor.getRed()) * intensity);
    int g = (int)(baseColor.getGreen() + (highlightColor.getGreen() -
baseColor.getGreen()) * intensity);
    int b = (int)(baseColor.getBlue() + (highlightColor.getBlue() -
baseColor.getBlue()) * intensity);
    return new Color(clamp(r), clamp(g), clamp(b));
}

// Null-safe version of add
public static Color addSafe(Color c1, Color c2) {
    if(c1 == null && c2 == null) return BLACK;
    if(c1 == null) return c2;
    if(c2 == null) return c1;
}

```

```

    return add(c1, c2);
}

/***
 * Clamps float value between [min, max]
 */
public static float clamp(float value, float min, float max) {
    return Math.max(min, Math.min(max, value));
}

public static double clampDouble(double value, double min, double max)
{
    return Math.max(min, Math.min(max, value));
}

/***
 * Bilinear interpolation between four colors
 * Result is always fully opaque (alpha=255)
 */
public static Color bilinearInterpolate(Color c00, Color c10, Color c01,
Color c11, double tx, double ty) {
    int r = (int)((1-tx)*(1-ty)*c00.getRed() + tx*(1-ty)*c10.getRed() +
(1-tx)*ty*c01.getRed() + tx*ty*c11.getRed());
    int g = (int)((1-tx)*(1-ty)*c00.getGreen() + tx*(1-ty)*c10.getGreen() +
(1-tx)*ty*c01.getGreen() + tx*ty*c11.getGreen());
    int b = (int)((1-tx)*(1-ty)*c00.getBlue() + tx*(1-ty)*c10.getBlue() +
(1-tx)*ty*c01.getBlue() + tx*ty*c11.getBlue());

    return new Color(clamp(r), clamp(g), clamp(b));
}

/***
 * Blends two colors with given ratio (0.0-1.0)
 * Result is always fully opaque (alpha=255)
 */
public static Color blendColors(Color color1, Color color2, float ratio) {
    ratio = clamp(ratio, 0.0f, 1.0f);
    int r = (int)(color1.getRed() * (1 - ratio) + color2.getRed() * ratio);
    int g = (int)(color1.getGreen() * (1 - ratio) + color2.getGreen() * ratio);

```

```

int b = (int)(color1.getBlue() * (1 - ratio) + color2.getBlue() * ratio);
return new Color(clamp(r), clamp(g), clamp(b));
}

// Double ratio version
public static Color blendColors(Color color1, Color color2, double ratio)
{
    ratio = Math.max(0, Math.min(1, ratio));
    int r = (int)(color1.getRed() * (1-ratio) + color2.getRed() * ratio);
    int g = (int)(color1.getGreen() * (1-ratio) + color2.getGreen() * ratio);
    int b = (int)(color1.getBlue() * (1-ratio) + color2.getBlue() * ratio);
    return new Color(clamp(r), clamp(g), clamp(b));
}

/***
 * Smooth blending using smoothstep function
 * Result is always fully opaque (alpha=255)
 */
public static Color smoothBlend(Color c1, Color c2, double ratio) {
    ratio = Math.max(0, Math.min(1, ratio));
    double smoothRatio = ratio * ratio * (3 - 2 * ratio);
    return blendColors(c1, c2, (float)smoothRatio);
}

/***
 * Calculates luminance (brightness) of color using ITU-R BT.709
standard
*/
public static double luminance(Color color) {
    return (0.2126 * color.getRed() + 0.7152 * color.getGreen() + 0.0722 *
color.getBlue()) / 255.0;
}

/***
 * Adjusts color contrast
 * Result is always fully opaque (alpha=255)
*/
public static Color adjustContrast(Color color, float contrast) {
    float factor = (259f * (contrast + 255f)) / (255f * (259f - contrast));
}

```

```

int red = adjustComponent(color.getRed(), factor);
int green = adjustComponent(color.getGreen(), factor);
int blue = adjustComponent(color.getBlue(), factor);
return new Color(clamp(red), clamp(green), clamp(blue));
}

private static int adjustComponent(int component, float factor) {
    float normalized = component / 255f;
    float adjusted = 0.5f + factor * (normalized - 0.5f);
    return (int)(adjusted * 255f);
}

/***
 * Adjusts color brightness (exposure)
 * Result is always fully opaque (alpha=255)
 */
public static Color adjustExposure(Color color, float exposure) {
    float[] rgb = getFloatComponents(color);
    return new Color(
        clamp(rgb[0] * exposure, 0f, 1f),
        clamp(rgb[1] * exposure, 0f, 1f),
        clamp(rgb[2] * exposure, 0f, 1f)
    );
}

/***
 * Adjusts color saturation
 * Result is always fully opaque (alpha=255)
 */
public static Color adjustSaturation(Color color, float saturation) {
    float[] rgb = getFloatComponents(color);
    float luminance = 0.2126f * rgb[0] + 0.7152f * rgb[1] + 0.0722f *
rgb[2];
    return new Color(
        clamp(luminance + (rgb[0] - luminance) * saturation, 0f, 1f),
        clamp(luminance + (rgb[1] - luminance) * saturation, 0f, 1f),
        clamp(luminance + (rgb[2] - luminance) * saturation, 0f, 1f)
    );
}

```

```

/***
 * Inverts color (negative)
 * Result is always fully opaque (alpha=255)
 */
public static Color invert(Color color) {
    return new Color(
        255 - color.getRed(),
        255 - color.getGreen(),
        255 - color.getBlue()
    );
}

/***
 * Shifts hue in HSV color space
 * Result is always fully opaque (alpha=255)
 */
public static Color shiftHue(Color color, float hueShift) {
    float[] hsb = Color.RGBtoHSB(color.getRed(), color.getGreen(),
        color.getBlue(), null);
    float newHue = (hsb[0] + hueShift/360f) % 1f;
    if (newHue < 0) newHue += 1f;
    return Color.getHSBColor(newHue, hsb[1], hsb[2]);
}

/***
 * Adjusts color temperature (warm/cool)
 * Result is always fully opaque (alpha=255)
 */
public static Color adjustTemperature(Color color, float temperature) {
    temperature = clamp(temperature, -1f, 1f);
    float[] rgb = getFloatComponents(color);
    if (temperature > 0) {
        rgb[0] += temperature;
        rgb[1] += temperature * 0.5f;
    } else {
        rgb[2] -= temperature;
    }
    return new Color(

```

```

        clamp(rgb[0], 0f, 1f),
        clamp(rgb[1], 0f, 1f),
        clamp(rgb[2], 0f, 1f)
    );
}

/***
 * Converts to black and white based on threshold
 * Result is always fully opaque (alpha=255)
 */
public static Color toBlackAndWhite(Color color, int threshold) {
    int luminance = (int)(luminance(color) * 255);
    return luminance > threshold ? WHITE : BLACK;
}

/***
 * Converts to sepia tone
 * Result is always fully opaque (alpha=255)
 */
public static Color toSepia(Color color) {
    int r = color.getRed();
    int g = color.getGreen();
    int b = color.getBlue();
    int tr = (int)(0.393 * r + 0.769 * g + 0.189 * b);
    int tg = (int)(0.349 * r + 0.686 * g + 0.168 * b);
    int tb = (int)(0.272 * r + 0.534 * g + 0.131 * b);
    return new Color(clamp(tr), clamp(tg), clamp(tb));
}

/***
 * Calculates Euclidean distance between two colors
 */
public static double colorDistance(Color c1, Color c2) {
    double rDiff = c1.getRed() - c2.getRed();
    double gDiff = c1.getGreen() - c2.getGreen();
    double bDiff = c1.getBlue() - c2.getBlue();
    return Math.sqrt(rDiff*rDiff + gDiff*gDiff + bDiff*bDiff);
}

```

```
/***
 * Sets new alpha value for existing color
 * This is the ONLY method that handles alpha - for compatibility
 */
public static Color setAlpha(Color color, int alpha) {
    alpha = clamp(alpha, 0, 255);
    return new Color(color.getRed(), color.getGreen(), color.getBlue(),
alpha);
}

/***
 * Sets new alpha value (float 0.0-1.0)
 * This is the ONLY method that handles alpha - for compatibility
 */
public static Color setAlpha(Color color, float alpha) {
    alpha = clamp(alpha, 0.0f, 1.0f);
    return new Color(
        color.getRed() / 255f,
        color.getGreen() / 255f,
        color.getBlue() / 255f,
        alpha
    );
}

/***
 * Clamp integer value to [0,255]
 */
public static int clamp(int value, int min, int max) {
    return Math.max(min, Math.min(max, value));
}

public static int clamp(int value) {
    return Math.max(0, Math.min(255, value));
}

public static double clampDoubleColorValue(double value) {
    return Math.max(0.0, Math.min(1.0, value));
}
```

```

/***
 * Scales color by factor (0.0-1.0)
 * Result is always fully opaque (alpha=255)
 */
public static Color scale(Color color, double factor) {
    factor = Math.max(0.0, Math.min(1.0, factor));
    int r = (int)(color.getRed() * factor);
    int g = (int)(color.getGreen() * factor);
    int b = (int)(color.getBlue() * factor);

    return new Color(clamp(r), clamp(g), clamp(b));
}

/***
 * Applies lighting model (diffuse only)
 * Result is always fully opaque (alpha=255)
 */
public static Color applyLighting(Color baseColor, Color lightColor,
double intensity, double NdotL) {
    int r = (int)(baseColor.getRed() * lightColor.getRed() / 255.0 * intensity
* NdotL);
    int g = (int)(baseColor.getGreen() * lightColor.getGreen() / 255.0 *
intensity * NdotL);
    int b = (int)(baseColor.getBlue() * lightColor.getBlue() / 255.0 *
intensity * NdotL);
    return new Color(clamp(r), clamp(g), clamp(b));
}

/***
 * Applies lighting with ambient and diffuse components
 * Result is always fully opaque (alpha=255)
 */
public static Color applyLightingX(Color base, Color light, double
diffuse, double ambient) {
    int r = (int)((base.getRed() * (ambient + diffuse *
light.getRed()/255.0)));
    int g = (int)((base.getGreen() * (ambient + diffuse *
light.getGreen()/255.0)));
    int b = (int)((base.getBlue() * (ambient + diffuse *

```

```
light.getBlue()/255.0));
    return new Color(clamp(r), clamp(g), clamp(b));
}

}

// =====
// File: /net/elenamurat/util/NoiseUtil.java
// =====

package net.elena.murat.util;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

public final class NoiseUtil {

    private static final int[] PERMUTATION = new int[512]; // 512-element
array

    static {
        // Base permutation table (0-255)
        int[] temp =
        { 151,160,137,91,90,15,131,13,201,95,96,53,194,233,7,225,
140,36,103,30,69,142,8,99,37,240,21,10,23,190,6,148,247,120,234,75,0,2
6,197,62,94,252,219,203,117,
35,11,32,57,177,33,88,237,149,56,87,174,20,125,136,171,168,68,175,74,1
65,71,134,139,48,27,166,77,
146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,
244,102,143,54,65,25,63,161,
1,216,80,73,209,76,132,187,208,89,18,169,200,196,135,130,116,188,159,
86,164,100,109,198,173,186,3,
64,52,217,226,250,124,123,5,202,38,147,118,126,255,82,85,212,207,206,
```

59,227,47,16,58,17,182,189,
28,42,223,183,170,213,119,248,152,2,44,154,163,70,221,153,101,155,167
,43,172,9,129,22,39,253,19,98,
108,110,79,113,224,232,178,185,112,104,218,246,97,228,251,34,242,193,
238,210,144,12,191,179,162,241,
81,51,145,235,249,14,239,107,49,192,214,31,181,199,106,157,184,84,204
,176,115,121,50,45,127,4,150,
254,138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,1
56,180 };

```
// Copy and repeat the array
System.arraycopy(temp, 0, PERMUTATION, 0, 256);
System.arraycopy(temp, 0, PERMUTATION, 256, 256);
}

/**
 * 3D Perlin noise (returns value between -1.0 and 1.0)
 */
public static double noise(Point3 point) {
    int xi = (int)Math.floor(point.x) & 255;
    int yi = (int)Math.floor(point.y) & 255;
    int zi = (int)Math.floor(point.z) & 255;

    double xf = point.x - Math.floor(point.x);
    double yf = point.y - Math.floor(point.y);
    double zf = point.z - Math.floor(point.z);

    double u = fade(xf);
    double v = fade(yf);
    double w = fade(zf);

    int aaa = PERMUTATION[PERMUTATION[PERMUTATION[xi] +
yi] + zi];
    int aba = PERMUTATION[PERMUTATION[PERMUTATION[xi] +
yi + 1] + zi];
```

```

int aab = PERMUTATION[PERMUTATION[PERMUTATION[xi] +
yi] + zi + 1];
int abb = PERMUTATION[PERMUTATION[PERMUTATION[xi] +
yi + 1] + zi + 1];
int baa = PERMUTATION[PERMUTATION[PERMUTATION[xi + 1]
+ yi] + zi];
int bba = PERMUTATION[PERMUTATION[PERMUTATION[xi + 1]
+ yi + 1] + zi];
int bab = PERMUTATION[PERMUTATION[PERMUTATION[xi + 1]
+ yi] + zi + 1];
int bbb = PERMUTATION[PERMUTATION[PERMUTATION[xi + 1]
+ yi + 1] + zi + 1];

double x1 = lerp(grad(aaa, xf, yf, zf), grad(baa, xf-1, yf, zf), u);
double x2 = lerp(grad(aba, xf, yf-1, zf), grad(bba, xf-1, yf-1, zf), u);
double y1 = lerp(x1, x2, v);

x1 = lerp(grad(aab, xf, yf, zf-1), grad(bab, xf-1, yf, zf-1), u);
x2 = lerp(grad(abb, xf, yf-1, zf-1), grad(bbb, xf-1, yf-1, zf-1), u);
double y2 = lerp(x1, x2, v);

return lerp(y1, y2, w);
}

/**
 * Turbulence effect (Fractal noise)
 * @param point 3D point
 * @param octaves Number of noise layers
 */
public static double turbulence(Point3 point, int octaves) {
    double value = 0.0;
    double size = 1.0;
    double totalAmplitude = 0.0;
    double amplitude = 1.0;

    for (int i = 0; i < octaves; i++) {
        value += amplitude * Math.abs(noise(new Point3(
            point.x / size,
            point.y / size,

```

```

        point.z / size
    )));
    totalAmplitude += amplitude;
    amplitude *= 0.5;
    size *= 0.5;
}

return value / totalAmplitude;
}

private static double fade(double t) {
    return t * t * t * (t * (t * 6 - 15) + 10);
}

private static double lerp(double a, double b, double t) {
    return a + t * (b - a);
}

private static double grad(int hash, double x, double y, double z) {
    int h = hash & 15;
    double u = h < 8 ? x : y;
    double v = h < 4 ? y : h == 12 || h == 14 ? x : z;
    return ((h & 1) == 0 ? u : -u) + ((h & 2) == 0 ? v : -v);
}

/***
 * Planar noise (2D)
 */
public static double noise(double x, double y) {
    return noise(new Point3(x, y, 0));
}

}

// =====
// File: /net/elenamurat/util/LetterUtils3D.java
// =====

```

```
package net.elena.murat.util;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;

public final class LetterUtils3D {
    private LetterUtils3D() {}

    // Cache mechanism with improved key generation
    private static final ConcurrentHashMap<String, LetterMesh>
    MESH_CACHE = new ConcurrentHashMap<>();

    public static BufferedImage getLetterImage(char c, Font font, double
widthScale, double heightScale, int size) {
        final double baseSize = (double)(size);
        int width = (int)(baseSize * widthScale);
        int height = (int)(baseSize * heightScale);

        BufferedImage img = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_ARGB);
        Graphics2D g = img.createGraphics();

        try {
            // Setup graphics
            g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
            g.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
            RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
            g.setBackground(new Color(0, 0, 0)); // Transparent background
            g.clearRect(0, 0, width, height);
            g.setColor(Color.BLACK);
            g.setFont(font);

            // Center the letter properly with correct orientation
            FontMetrics fm = g.getFontMetrics();
            int xPos = (width - fm.charWidth(c)) / 2;
```

```

int yPos = (height - fm.getHeight()) / 2 + fm.getAscent();

// Flip Y-axis to match 3D coordinate system
// g.translate(0, height);
// g.scale(1, -1);

g.drawString(String.valueOf(c), xPos, yPos);
} finally {
g.dispose();
}

return img;
}

public static boolean[][] getLetterPixelData(BufferedImage img) {
int width = img.getWidth();
int height = img.getHeight();
boolean[][] pixels = new boolean[width][height];
int[] rgb = new int[width * height];

img.getRGB(0, 0, width, height, rgb, 0, width);

final int rgblen=rgb.length;

// Read pixels with proper Y orientation (flipped vertically)
for (int i = 0; i < rgblen; i++) {
int x = i % width;
int y = height - 1 - (i / width); // Flip Y coordinate
pixels[x][y] = (rgb[i] & 0xFF000000) != 0; // Check alpha channel
}

return pixels;
}

public static LetterMesh getLetterMeshData(boolean[][] pixels, double
thickness) {
int width = pixels.length;
int height = pixels[0].length;

```

```

String cacheKey = width + "x" + height + "t" + thickness;
return MESH_CACHE.computeIfAbsent(cacheKey, k -> {
    List<Vertex> vertices = new ArrayList<>();
    List<Face> faces = new ArrayList<>();
    double halfThickness = thickness / 2;
    double scaleX = 1.0 / width;
    double scaleY = 1.0 / height;

    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            if (pixels[x][y]) {
                addOptimizedVoxel(vertices, faces, x, y, halfThickness, scaleX,
scaleY);
            }
        }
    }

    return new LetterMesh(vertices, faces);
});

private static void addOptimizedVoxel(List<Vertex> vertices,
List<Face> faces,
int x, int y, double halfThick, double scaleX, double scaleY) {
    double px = x * scaleX;
    double py = y * scaleY;
    int baseIndex = vertices.size();

    // Create 8 vertices for the voxel
    vertices.add(new Vertex(px,      py,      -halfThick)); // 0
    vertices.add(new Vertex(px + scaleX, py,      -halfThick)); // 1
    vertices.add(new Vertex(px + scaleX, py + scaleY, -halfThick)); // 2
    vertices.add(new Vertex(px,      py + scaleY, -halfThick)); // 3
    vertices.add(new Vertex(px,      py,      halfThick)); // 4
    vertices.add(new Vertex(px + scaleX, py,      halfThick)); // 5
    vertices.add(new Vertex(px + scaleX, py + scaleY, halfThick)); // 6
    vertices.add(new Vertex(px,      py + scaleY, halfThick)); // 7

    // Create 12 triangular faces (2 per cube face)
}

```

```

int[] faceIndices = {
    // Front face
    baseIndex, baseIndex+1, baseIndex+2,
    baseIndex, baseIndex+2, baseIndex+3,
    // Back face
    baseIndex+4, baseIndex+6, baseIndex+5,
    baseIndex+4, baseIndex+7, baseIndex+6,
    // Top face
    baseIndex, baseIndex+4, baseIndex+5,
    baseIndex, baseIndex+5, baseIndex+1,
    // Bottom face
    baseIndex+3, baseIndex+2, baseIndex+6,
    baseIndex+3, baseIndex+6, baseIndex+7,
    // Right face
    baseIndex+1, baseIndex+5, baseIndex+6,
    baseIndex+1, baseIndex+6, baseIndex+2,
    // Left face
    baseIndex, baseIndex+3, baseIndex+7,
    baseIndex, baseIndex+7, baseIndex+4
};

for (int i = 0; i < faceIndices.length; i += 3) {
    faces.add(new Face(
        faceIndices[i],
        faceIndices[i+1],
        faceIndices[i+2]
    ));
}

// Immutable vertex class
public static final class Vertex {
    public final double x, y, z;
    public Vertex(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}

```

```
// Immutable face class
public static final class Face {
    public final int v1, v2, v3;
    public Face(int v1, int v2, int v3) {
        this.v1 = v1;
        this.v2 = v2;
        this.v3 = v3;
    }
}

// Immutable mesh container
public static final class LetterMesh {
    public final List<Vertex> vertices;
    public final List<Face> faces;

    public LetterMesh(List<Vertex> vertices, List<Face> faces) {
        this.vertices = new ArrayList<>(vertices); // Java 6/8 uyumlu defensive
copy
        this.faces = new ArrayList<>(faces);      // Java 6/8 uyumlu defensive
copy
    }
}

}
```

```
// =====
// File: /net/elenamurat/util/ImageUtils3D.java
// =====
```

```
package net.elena.murat.util;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ConcurrentHashMap;
```

```
public final class ImageUtils3D {  
    private ImageUtils3D() {}  
  
    // Cache mechanism with improved key generation  
    private static final ConcurrentHashMap<String, ImageMesh>  
MESH_CACHE = new ConcurrentHashMap<>();  
  
    public static BufferedImage getBufferedImage(BufferedImage source,  
double widthScale, double heightScale, int size) {  
        final double baseSize = (double)(size);  
        int width = (int)(baseSize * widthScale);  
        int height = (int)(baseSize * heightScale);  
  
        BufferedImage img = new BufferedImage(width, height,  
BufferedImage.TYPE_INT_ARGB);  
        Graphics2D g = img.createGraphics();  
  
        try {  
            // Setup graphics  
            g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  
RenderingHints.VALUE_ANTIALIAS_ON);  
            g.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,  
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);  
            g.setBackground(new Color(0, 0, 0, 0)); // Transparent background  
            g.clearRect(0, 0, width, height);  
  
            g.drawImage(source, 0, 0, width, height, null);  
        } finally {  
            g.dispose();  
        }  
  
        return img;  
    }  
  
    public static boolean[][] getImagePixelData(BufferedImage img) {  
        int width = img.getWidth();  
        int height = img.getHeight();  
        boolean[][] pixels = new boolean[width][height];  
        int[] rgb = new int[width * height];
```

```
img.getRGB(0, 0, width, height, rgb, 0, width);

final int rgblen=rgb.length;

// Read pixels with proper Y orientation (flipped vertically)
for (int i = 0; i < rgblen; i++) {
    int x = i % width;
    int y = height - 1 - (i / width); // Flip Y coordinate
    pixels[x][y] = (rgb[i] & 0xFF000000) != 0; // Check alpha channel
}

return pixels;
}

public static BufferedImage convertToTransparentImage(BufferedImage srcImage, double transparency) {
    if ( (transparency <= 0.0) || (transparency >= 1.0) ) {
        return srcImage;
    }

    final int width = srcImage.getWidth();
    final int height = srcImage.getHeight();

    BufferedImage result = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_RGB);
    Graphics2D g = result.createGraphics();

    g.setBackground(Color.WHITE);
    g.clearRect (0, 0, width, height);

    // Apply transparency using AlphaComposite

    g.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
    (float)(1.0 - transparency)));
    g.drawImage(srcImage, 0, 0, null);

    //g.dispose();
```

```

System.out.println ("RESULT IMAGE TYPE: "+result.getType ()+"");
return result;
}

public static ImageMesh getImageMeshData(boolean[][] pixels, double
thickness) {
    int width = pixels.length;
    int height = pixels[0].length;

    String cacheKey = width + "x" + height + "t" + thickness;
    return MESH_CACHE.computeIfAbsent(cacheKey, k -> {
        List<Vertex> vertices = new ArrayList<>();
        List<Face> faces = new ArrayList<>();
        double halfThickness = thickness / 2;
        double scaleX = 1.0 / width;
        double scaleY = 1.0 / height;

        for (int x = 0; x < width; x++) {
            for (int y = 0; y < height; y++) {
                if (pixels[x][y]) {
                    addOptimizedVoxel(vertices, faces, x, y, halfThickness, scaleX,
scaleY);
                }
            }
        }
    });

    return new ImageMesh(vertices, faces);
}
}

private static void addOptimizedVoxel(List<Vertex> vertices,
List<Face> faces,
int x, int y, double halfThick, double scaleX, double scaleY) {
    double px = x * scaleX;
    double py = y * scaleY;
    int baseIndex = vertices.size();

```

```

// Create 8 vertices for the voxel
vertices.add(new Vertex(px,      py,      -halfThick)); // 0
vertices.add(new Vertex(px + scaleX, py,      -halfThick)); // 1
vertices.add(new Vertex(px + scaleX, py + scaleY, -halfThick)); // 2
vertices.add(new Vertex(px,      py + scaleY, -halfThick)); // 3
vertices.add(new Vertex(px,      py,      halfThick)); // 4
vertices.add(new Vertex(px + scaleX, py,      halfThick)); // 5
vertices.add(new Vertex(px + scaleX, py + scaleY, halfThick)); // 6
vertices.add(new Vertex(px,      py + scaleY, halfThick)); // 7

// Create 12 triangular faces (2 per cube face)
int[] faceIndices = {
    // Front face
    baseIndex, baseIndex+1, baseIndex+2,
    baseIndex, baseIndex+2, baseIndex+3,
    // Back face
    baseIndex+4, baseIndex+6, baseIndex+5,
    baseIndex+4, baseIndex+7, baseIndex+6,
    // Top face
    baseIndex, baseIndex+4, baseIndex+5,
    baseIndex, baseIndex+5, baseIndex+1,
    // Bottom face
    baseIndex+3, baseIndex+2, baseIndex+6,
    baseIndex+3, baseIndex+6, baseIndex+7,
    // Right face
    baseIndex+1, baseIndex+5, baseIndex+6,
    baseIndex+1, baseIndex+6, baseIndex+2,
    // Left face
    baseIndex, baseIndex+3, baseIndex+7,
    baseIndex, baseIndex+7, baseIndex+4
};

for (int i = 0; i < faceIndices.length; i += 3) {
    faces.add(new Face(
        faceIndices[i],
        faceIndices[i+1],
        faceIndices[i+2]
    ));
}

```

```
}

// Immutable vertex class
public static final class Vertex {
    public final double x, y, z;
    public Vertex(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }
}

// Immutable face class
public static final class Face {
    public final int v1, v2, v3;
    public Face(int v1, int v2, int v3) {
        this.v1 = v1;
        this.v2 = v2;
        this.v3 = v3;
    }
}

// Immutable mesh container
public static final class ImageMesh {
    public final List<Vertex> vertices;
    public final List<Face> faces;

    public ImageMesh(List<Vertex> vertices, List<Face> faces) {
        this.vertices = new ArrayList<>(vertices); // Java 6/8 uyumlu defensive
copy
        this.faces = new ArrayList<>(faces);      // Java 6/8 uyumlu defensive
copy
    }
}

// =====
```

```
// File: /net/elenamurat/util/ResizeImage.java
// =====

package net.elena.murat.util;

import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.image.BufferedImage;

/**
 * Utility class for resizing BufferedImages while maintaining aspect ratio.
 */
public class ResizeImage {

    /**
     * Resizes a given BufferedImage to new dimensions (newWidth, newHeight)
     * while preserving its aspect ratio. The image will be scaled to fit
     * within the new dimensions and centered, potentially adding padding
     * (letterboxing or pillarboxing) if the aspect ratios differ.
     *
     * @param src The source BufferedImage to be resized.
     * @param newWidth The desired width for the resized image.
     * @param newHeight The desired height for the resized image.
     * @param backgroundColor The background color for padding areas.
     * Pass null for transparent background (if image type supports it).
     * @return A new BufferedImage with the specified dimensions,
     * containing the scaled source image.
     */
    public static BufferedImage getResizedImage(BufferedImage src, int
newWidth, int newHeight, Color backgroundColor) {
        // Determine the type for the new buffered image.
        // Use TYPE_INT_ARGB if a transparent background is desired
        // (backgroundColor is null),
        // otherwise TYPE_INT_RGB for opaque images.
        int imageType = BufferedImage.TYPE_INT_RGB;
        if (backgroundColor == null) {
            imageType = BufferedImage.TYPE_INT_ARGB;
```

```
}
```

```
    BufferedImage resizedImage = new BufferedImage(newWidth,  
newHeight, imageType);  
    Graphics2D g2d = resizedImage.createGraphics();  
  
    // Set rendering hints for high-quality scaling  
    g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,  
RenderingHints.VALUE_INTERPOLATION_BILINEAR);  
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING,  
RenderingHints.VALUE_RENDER_QUALITY);  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  
RenderingHints.VALUE_ANTIALIAS_ON);  
  
    // Fill background if a color is provided  
    if (backgroundColor != null) {  
        g2d.setColor(backgroundColor);  
        g2d.fillRect(0, 0, newWidth, newHeight);  
    }  
  
    // Calculate scaling factors and drawing dimensions to preserve aspect  
ratio  
    double originalWidth = src.getWidth();  
    double originalHeight = src.getHeight();  
  
    double scaleX = newWidth / originalWidth;  
    double scaleY = newHeight / originalHeight;  
  
    // Use the smaller scale factor to ensure the entire image fits within the  
new bounds  
    double scale = Math.min(scaleX, scaleY);  
  
    int scaledWidth = (int) (originalWidth * scale);  
    int scaledHeight = (int) (originalHeight * scale);  
  
    // Calculate position to center the scaled image on the new canvas  
    int x = (newWidth - scaledWidth) / 2;  
    int y = (newHeight - scaledHeight) / 2;
```

```

// Draw the scaled image onto the new buffered image
g2d.drawImage(src, x, y, scaledWidth, scaledHeight, null);
g2d.dispose(); // Release Graphics2D resources

return resizedImage;
}

/***
 * Overload for getResizedImage that defaults to a black background for
padding.
 * @param src The source BufferedImage to be resized.
 * @param newWidth The desired width for the resized image.
 * @param newHeight The desired height for the resized image.
 * @return A new BufferedImage with the specified dimensions,
containing the scaled source image.
*/
public static BufferedImage getResizedImage(BufferedImage src, int
newWidth, int newHeight) {
    return getResizedImage(src, newWidth, newHeight, Color.BLACK); // Default to black background
}
}

```

```

// =====
// File: /net/elenamurat/util/MathUtil.java
// =====

```

```

package net.elenamurat.util;

import java.util.Random;

import net.elenamurat.math.Vector3;
import net.elenamurat.math.Point3;

/***
 * Mathematical helper functions for 3D graphics and ray tracing.
 * All methods are thread-safe and deterministic.
*/

```

```
public final class MathUtil {  
  
    // Mathematical constants  
    public static final double PI = Math.PI;  
    public static final double TWO_PI = 2.0 * PI;  
    public static final double INV_PI = 1.0 / PI;  
    public static final double EPSILON = 1e-8;  
    public static final double GOLDEN_RATIO = 1.618033988749895;  
  
    private static final Random RAND = new Random();  
    private static final int[] PRIMES = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};  
  
    private MathUtil() {} // Prevent instantiation  
  
    // --- Basic Math Functions ---  
  
    /**  
     * Clamps value between [min, max] range.  
     */  
    public static double clamp(double value, double min, double max) {  
        return Math.max(min, Math.min(max, value));  
    }  
  
    /**  
     * Converts degrees to radians.  
     */  
    public static double radians(double degrees) {  
        return degrees * PI / 180.0;  
    }  
  
    /**  
     * Converts radians to degrees.  
     */  
    public static double degrees(double radians) {  
        return radians * 180.0 / PI;  
    }  
  
    /**  
     * Linear interpolation (lerp).  
    */
```

```

*/
public static double lerp(double a, double b, double t) {
    return a + t * (b - a);
}

// --- Random Number Generation ---

/***
 * Deterministic random number in [0,1) range.
 * @param seed Complex seed value
 */
public static double random(double seed) {
    double x = Math.sin(seed * 12.9898 + 78.233) * 43758.5453;
    return x - Math.floor(x);
}

/***
 * Random point inside 3D unit sphere.
 */
public static Vector3 randomInUnitSphere(double seed) {
    double phi = TWO_PI * random(seed + PRIMES[0]);
    double costheta = 2.0 * random(seed + PRIMES[1]) - 1.0;
    double theta = Math.acos(costheta);
    double r = Math.cbrt(random(seed + PRIMES[2]));

    return new Vector3(
        r * Math.sin(theta) * Math.cos(phi),
        r * Math.sin(theta) * Math.sin(phi),
        r * Math.cos(theta)
    );
}

// --- Vector and Point Operations ---

/***
 * Calculates ray-plane intersection.
 * @param rayOrigin Ray starting point
 * @param rayDir Ray direction (normalized)
 * @param planePoint Point on the plane

```

```

* @param planeNormal Plane normal (normalized)
* @return Intersection distance (Double.POSITIVE_INFINITY if
parallel)
*/
public static double rayPlaneIntersect(
    Point3 rayOrigin, Vector3 rayDir,
    Point3 planePoint, Vector3 planeNormal) {

    double denom = planeNormal.dot(rayDir);
    if (Math.abs(denom) > EPSILON) {
        Vector3 diff = planePoint.subtract(rayOrigin);
        return diff.dot(planeNormal) / denom;
    }
    return Double.POSITIVE_INFINITY;
}

/**
 * Ray-triangle intersection (Möller-Trumbore algorithm).
*/
public static Double rayTriangleIntersect(
    Point3 rayOrigin, Vector3 rayDir,
    Point3 v0, Point3 v1, Point3 v2) {

    Vector3 edge1 = v1.subtract(v0);
    Vector3 edge2 = v2.subtract(v0);
    Vector3 h = rayDir.cross(edge2);
    double a = edge1.dot(h);

    if (a > -EPSILON && a < EPSILON) {
        return null; // Ray parallel to plane
    }

    double f = 1.0 / a;
    Vector3 s = rayOrigin.subtract(v0);
    double u = f * s.dot(h);

    if (u < 0.0 || u > 1.0) {
        return null;
    }
}

```

```

Vector3 q = s.cross(edge1);
double v = f * rayDir.dot(q);

if (v < 0.0 || u + v > 1.0) {
    return null;
}

double t = f * edge2.dot(q);
return t > EPSILON ? t : null;
}

// --- Noise Functions ---

/***
 * Hash function for Perlin noise.
 */
public static int noiseHash(int x, int y, int z) {
    final int X_NOISE = 1619;
    final int Y_NOISE = 31337;
    final int Z_NOISE = 6971;
    final int SEED = 1013;

    int hash = (x * X_NOISE) ^ (y * Y_NOISE) ^ (z * Z_NOISE);
    hash = hash * hash * hash * SEED;
    return (hash >> 13) ^ hash;
}

/***
 * 3D Perlin noise (between -1 and 1).
 */
public static double perlinNoise(double x, double y, double z) {
    // Simplified Perlin noise implementation
    int xi = (int)Math.floor(x) & 255;
    int yi = (int)Math.floor(y) & 255;
    int zi = (int)Math.floor(z) & 255;

    double xf = x - Math.floor(x);
    double yf = y - Math.floor(y);

```

```

double zf = z - Math.floor(z);

// Actual Perlin noise calculation would go here
// For simplicity, returning random value
return random(xi + yi * 256 + zi * 65536) * 2 - 1;
}

// --- Special Mathematical Functions ---

/***
 * Filters near-zero values.
 */
public static double nearZero(double value) {
    return Math.abs(value) < EPSILON ? 0.0 : value;
}

/***
 * Fresnel equation (Schlick approximation).
 * @param cosTheta Cosine of incident angle
 * @param refIdx Refractive index
 */
public static double fresnelSchlick(double cosTheta, double refIdx) {
    double r0 = (1 - refIdx) / (1 + refIdx);
    r0 = r0 * r0;
    return r0 + (1 - r0) * Math.pow(1 - cosTheta, 5);
}

/***
 * GGX distribution function (PBR specular).
 */
public static double ggxDistribution(double NdotH, double roughness) {
    double a = roughness * roughness;
    double a2 = a * a;
    double denom = (NdotH * NdotH * (a2 - 1.0) + 1.0);
    return a2 / (PI * denom * denom);
}

/***
 * Smith shadowing function.

```

```

*/
public static double smithG1(double NdotV, double roughness) {
    double k = (roughness + 1.0) * (roughness + 1.0) / 8.0;
    return NdotV / (NdotV * (1.0 - k) + k);
}

// --- Coordinate Transformations ---

/***
 * Spherical to Cartesian coordinate conversion.
 */
public static Vector3 sphericalToCartesian(double r, double theta, double phi) {
    double sinTheta = Math.sin(theta);
    return new Vector3(
        r * sinTheta * Math.cos(phi),
        r * sinTheta * Math.sin(phi),
        r * Math.cos(theta)
    );
}

/***
 * Cartesian to spherical coordinate conversion.
 */
public static double[] cartesianToSpherical(Vector3 v) {
    double r = v.length();
    return new double[] {
        r,
        Math.acos(v.z / r),
        Math.atan2(v.y, v.x)
    };
}

// =====
// File: /net/elenamurat/util/MaterialUtils.java
// =====

```

```
package net.elena.murat.util;

import java.awt.Color;

/**
 * Utility class for common material-related operations,
 * such as color manipulation.
 */
public class MaterialUtils {

    /**
     * Multiplies a Color by a scalar factor.
     * Each RGB component is multiplied by the factor and clamped to the
     * [0, 255] range.
     *
     * @param color The original Color to multiply.
     * @param factor The scalar factor to multiply by.
     * @return A new Color object resulting from the multiplication.
     */
    public static Color multiply(Color color, double factor) {
        int r = (int) (color.getRed() * factor);
        int g = (int) (color.getGreen() * factor);
        int b = (int) (color.getBlue() * factor);

        // Clamp values to the valid [0, 255] range
        r = Math.min(255, Math.max(0, r));
        g = Math.min(255, Math.max(0, g));
        b = Math.min(255, Math.max(0, b));

        return new Color(r, g, b);
    }

    /**
     * Adds two Color objects component-wise.
     * Each RGB component is added and clamped to the [0, 255] range.
     *
     * @param color1 The first Color.
     * @param color2 The second Color.
     */
}
```

```

* @return A new Color object resulting from the addition.
*/
public static Color add(Color color1, Color color2) {
    int r = color1.getRed() + color2.getRed();
    int g = color1.getGreen() + color2.getGreen();
    int b = color1.getBlue() + color2.getBlue();

    // Clamp values to the valid [0, 255] range
    r = Math.min(255, Math.max(0, r));
    g = Math.min(255, Math.max(0, g));
    b = Math.min(255, Math.max(0, b));

    return new Color(r, g, b);
}

/***
 * Multiplies two Color objects component-wise (e.g., for texture
blending).
 * Each RGB component is multiplied, normalized to [0, 1] for
multiplication,
 * then scaled back to [0, 255] and clamped.
 *
 * @param color1 The first Color.
 * @param color2 The second Color.
 * @return A new Color object resulting from the component-wise
multiplication.
*/
public static Color blend(Color color1, Color color2) {
    double r = (color1.getRed() / 255.0) * (color2.getRed() / 255.0);
    double g = (color1.getGreen() / 255.0) * (color2.getGreen() / 255.0);
    double b = (color1.getBlue() / 255.0) * (color2.getBlue() / 255.0);

    int finalR = (int) (r * 255.0);
    int finalG = (int) (g * 255.0);
    int finalB = (int) (b * 255.0);

    // Clamp values to the valid [0, 255] range
    finalR = Math.min(255, Math.max(0, finalR));
    finalG = Math.min(255, Math.max(0, finalG));

```

```

finalB = Math.min(255, Math.max(0, finalB));

    return new Color(finalR, finalG, finalB);
}

}

// =====
// File: /net/elenamurat/math/PolynomialSolver.java
// =====

package net.elena.murat.math;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class PolynomialSolver {
    public static final double EPS = 1e-10; // Tighter tolerance
    private static final double CUBIC_EPS = 1e-8; // Special tolerance for
cubic equations

    // Helper method: Filters numbers close to zero
    private static boolean isApproxZero(double val, double epsilon) {
        return Math.abs(val) < epsilon;
    }

    // Linear equation solver ( $ax + b = 0$ )
    public static List<Double> solveLinear(double a, double b) {
        if (isApproxZero(a, EPS)) {
            return isApproxZero(b, EPS) ?
Collections.singletonList(Double.POSITIVE_INFINITY) : // Infinite
solutions
                Collections.emptyList();
        }
        return Collections.singletonList(-b / a);
    }
}

```

```

}

// Quadratic equation solver (ax2 + bx + c = 0)
public static List<Double> solveQuadratic(double a, double b, double c)
{
    if (isApproxZero(a, EPS)) return solveLinear(b, c);

    double discriminant = b*b - 4*a*c;
    List<Double> roots = new ArrayList<>();

    if (discriminant < -EPS) return roots;

    if (isApproxZero(discriminant, EPS)) {
        roots.add(-b / (2*a));
    } else {
        double sqrtDisc = Math.sqrt(discriminant);
        roots.add((-b + sqrtDisc) / (2*a));
        roots.add((-b - sqrtDisc) / (2*a));
    }
    return roots;
}

// Cubic equation solver (x3 + a2x2 + a1x + a0 = 0)
public static List<Double> solveCubic(double a2, double a1, double a0)
{
    // Convert to depressed form: y3 + py + q = 0
    double p = a1 - a2*a2/3.0;
    double q = a0 - a2*a1/3.0 + 2*a2*a2*a2/27.0;

    // Special case: p ≈ 0
    if (isApproxZero(p, CUBIC_EPS)) {
        return solveLinear(1.0, q).stream()
            .map(y -> y - a2/3.0)
            .collect(Collectors.toList());
    }

    double discriminant = q*q/4.0 + p*p*p/27.0;
    List<Double> roots = new ArrayList<>();
    double offset = a2 / 3.0;
}

```

```

if (discriminant > EPS) { // 1 real root
    double u = cbrt(-q/2.0 + Math.sqrt(discriminant));
    double v = cbrt(-q/2.0 - Math.sqrt(discriminant));
    roots.add(u + v - offset);
}
else if (discriminant < -EPS) { // 3 real roots
    double angle = Math.acos(3*q/(2*p)*Math.sqrt(-3/p));
    for (int k = 0; k < 3; k++) {
        roots.add(2*Math.sqrt(-p/3.0) *
            Math.cos((angle - 2*k*Math.PI)/3.0) - offset);
    }
}
else { // Coincident roots
    double root = cbrt(q/2.0) - offset;
    Collections.addAll(roots, root, root, root);
}

return roots.stream().distinct().collect(Collectors.toList());
}

```

```

public static List<Double> solveQuartic(double a3, double a2, double a1,
double a0) {
    double p = a2 - 3*a3*a3/8.0;
    double q = a1 - a2*a3/2.0 + a3*a3*a3/8.0;
    double r = a0 - a1*a3/4.0 + a2*a3*a3/16.0 - 3*a3*a3*a3*a3/256.0;

    // Biquadratic case ( $q \approx 0$ )
    if (isApproxZero(q, EPS*10)) {
        List<Double> roots = solveQuadratic(1.0, p, r).stream()
            .filter(z -> z >= -EPS)
            .flatMap(z -> {
                double sqrtZ = Math.sqrt(z);
                return isApproxZero(sqrtZ, EPS) ?
                    Collections.singletonList(sqrtZ).stream() :
                    Stream.of(sqrtZ, -sqrtZ);
            })
            .collect(Collectors.toList());
    }
}

```

```

        return roots.stream()
            .map(y -> y - a3/4.0)
            .collect(Collectors.toList());
    }

    // Ferrari's method
    List<Double> cubicRoots = solveCubic(
        2*p,
        p*p - 4*r,
        -q*q
    ).stream()
        .filter(z -> z >= -EPS)
        .collect(Collectors.toList());

    if (cubicRoots.isEmpty()) return Collections.emptyList();

    double z = cubicRoots.get(0);
    double sqrt2z = Math.sqrt(2*z);

    // Two quadratic equations
    List<Double> roots = new ArrayList<>();
    double[] params = {
        sqrt2z, z + p/2.0 + q/(2*sqrt2z),
        -sqrt2z, z + p/2.0 - q/(2*sqrt2z)
    };

    for (int i = 0; i < 2; i++) {
        solveQuadratic(1.0, params[2*i], params[2*i+1]).stream()
            .map(y -> y - a3/4.0)
            .forEach(roots::add);
    }

    return roots.stream()
        .filter(t -> !Double.isNaN(t))
        .distinct()
        .collect(Collectors.toList());
}

// Cube root calculation (sign-preserving)

```

```

private static double cbrt(double x) {
    return x < 0 ? -Math.pow(-x, 1.0/3.0) : Math.pow(x, 1.0/3.0);
}

}

// =====
// File: /net/elenamurat/math/IntersectionInterval.java
// =====

package net.elena.murat.math;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

/**
 * Represents an interval where a ray is inside a shape.
 * Contains both entry (in) and exit (out) intersection information.
 * Used for CSG (Constructive Solid Geometry) operations.
 */
public class IntersectionInterval {
    public final double tIn, tOut;
    public final Intersection in, out;

    /**
     * Constructs an interval with entry and exit data.
     * @param tIn The parameter t where the ray enters the shape.
     * @param tOut The parameter t where the ray exits the shape.
     * @param in The Intersection at the entry point.
     * @param out The Intersection at the exit point.
     */
    public IntersectionInterval(double tIn, double tOut, Intersection in,
                               Intersection out) {
        this.tIn = tIn;
        this.tOut = tOut;
        this.in = in;
    }
}

```

```

        this.out = out;
    }

/***
 * Creates a degenerate interval for non-solid shapes (e.g., planes).
 * @param t The intersection parameter.
 * @param hit The intersection data (used for both in and out).
 * @return A new IntersectionInterval with tIn = tOut.
 */
public static IntersectionInterval point(double t, Intersection hit) {
    return new IntersectionInterval(t, t, hit, hit);
}

/***
 * Returns the t values (tIn, tOut) in sorted order (ascending).
 * Useful for CSG boundary analysis.
 * @return Unmodifiable list of doubles in ascending order.
 */
public List<Double> getTSorted() {
    if (tIn <= tOut) {
        return Arrays.asList(tIn, tOut);
    } else {
        return Arrays.asList(tOut, tIn);
    }
}

@Override
public String toString() {
    return String.format("IntersectionInterval(tIn=%.4f, tOut=%.4f, shape=%s",
            tIn, tOut, in != null && in.getShape() != null ?
            in.getShape().getClass().getSimpleName() : "null");
}

// =====
// File: /net/elenamurat/math/Point3.java

```

```
// ======

package net.elena.murat.math;

public class Point3 {
    public final double x;
    public final double y;
    public final double z;

    public static final Point3 ORIGIN = new Point3(0, 0, 0);

    public Point3(double x, double y, double z) {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public double get(int axis) {
        switch(axis) {
            case 0: return x;
            case 1: return y;
            case 2: return z;
            default: throw new IllegalArgumentException("Axis must be 0, 1 or
2");
        }
    }

    public Point3 add(Vector3 v) {
        return new Point3(x + v.x, y + v.y, z + v.z);
    }

    public Vector3 subtract(Point3 other) {
        return new Vector3(x - other.x, y - other.y, z - other.z);
    }

    public Point3 subtract(Vector3 v) {
        return new Point3(this.x - v.x, this.y - v.y, this.z - v.z);
    }
}
```

```

public Vector3 toVector() {
    return new Vector3(x, y, z);
}

public Vector3 toVector3() {
    return new Vector3(x, y, z);
}

public Vector3 multiply(double scalar) {
    return new Vector3(
        this.x * scalar,
        this.y * scalar,
        this.z * scalar
    );
}

public double length() {
    return Math.sqrt(x * x + y * y + z * z);
}

/**
 * Calculates the Euclidean distance between this point and another
Point3.
 * @param other The other point.
 * @return The distance between the two points.
 */
public double distance(Point3 other) {
    double dx = this.x - other.x;
    double dy = this.y - other.y;
    double dz = this.z - other.z;
    return Math.sqrt(dx * dx + dy * dy + dz * dz);
}

/**
 * Calculates the dot product between this point and another point/vector.
 * @param other The other point or vector
 * @return The dot product (scalar value)
 */
public double dot(Point3 other) {

```

```

        return this.x * other.x + this.y * other.y + this.z * other.z;
    }

/***
 * Calculates the dot product between this point and a vector.
 * @param vector The vector to calculate dot product with
 * @return The dot product (scalar value)
 */
public double dot(Vector3 vector) {
    return this.x * vector.x + this.y * vector.y + this.z * vector.z;
}

@Override
public String toString() {
    return String.format("Point3(%f, %f, %f)", x, y, z);
}
}

```

```

// =====
// File: /net/elenamurat/math/Vector2.java
// =====

```

```

package net.elena.murat.math;

/***
 * 2D vector class for graphics and mathematical operations.
 * Supports basic vector operations and transformations.
 */
public class Vector2 {
    public final double x;
    public final double y;

    /**
     * Constructor with x and y components
     * @param x X component
     * @param y Y component
     */
    public Vector2(double x, double y) {

```

```
    this.x = x;
    this.y = y;
}

/***
 * Default constructor (creates zero vector)
 */
public Vector2() {
    this(0.0, 0.0);
}

/***
 * Copy constructor
 * @param other Vector to copy
 */
public Vector2(Vector2 other) {
    this(other.x, other.y);
}

// --- Basic Arithmetic Operations ---

/***
 * Vector addition
 * @param other Vector to add
 * @return New vector representing the sum
 */
public Vector2 add(Vector2 other) {
    return new Vector2(this.x + other.x, this.y + other.y);
}

/***
 * Vector subtraction
 * @param other Vector to subtract
 * @return New vector representing the difference
 */
public Vector2 subtract(Vector2 other) {
    return new Vector2(this.x - other.x, this.y - other.y);
}
```

```

/**
 * Scalar multiplication
 * @param scalar Scalar value to multiply
 * @return New scaled vector
 */
public Vector2 multiply(double scalar) {
    return new Vector2(this.x * scalar, this.y * scalar);
}

/**
 * Scalar division
 * @param scalar Scalar value to divide by
 * @return New divided vector
 * @throws ArithmeticException if scalar is zero
 */
public Vector2 divide(double scalar) {
    if (Math.abs(scalar) < 1e-10) {
        throw new ArithmeticException("Division by zero");
    }
    return new Vector2(this.x / scalar, this.y / scalar);
}

/**
 * Component-wise multiplication
 * @param other Vector to multiply with
 * @return New vector with component-wise product
 */
public Vector2 multiply(Vector2 other) {
    return new Vector2(this.x * other.x, this.y * other.y);
}

/**
 * Component-wise division
 * @param other Vector to divide by
 * @return New vector with component-wise quotient
 * @throws ArithmeticException if any component of other is zero
 */
public Vector2 divide(Vector2 other) {
    if (Math.abs(other.x) < 1e-10 || Math.abs(other.y) < 1e-10) {

```

```
        throw new ArithmeticException("Division by zero component");
    }
    return new Vector2(this.x / other.x, this.y / other.y);
}

// --- Vector Operations ---

/***
 * Dot product of two vectors
 * @param other Other vector
 * @return Dot product value
 */
public double dot(Vector2 other) {
    return this.x * other.x + this.y * other.y;
}

/***
 * Calculates the magnitude (length) of the vector
 * @return Magnitude of the vector
 */
public double length() {
    return Math.sqrt(x * x + y * y);
}

/***
 * Calculates the squared magnitude of the vector
 * (faster than length() for comparison purposes)
 * @return Squared magnitude
 */
public double lengthSquared() {
    return x * x + y * y;
}

/***
 * Normalizes the vector (makes it unit length)
 * @return New normalized vector
 * @throws ArithmeticException if vector length is zero
 */
public Vector2 normalize() {
```

```

        double len = length();
        if (len < 1e-10) {
            throw new ArithmeticException("Cannot normalize zero vector");
        }
        return new Vector2(x / len, y / len);
    }

    /**
     * Calculates the distance between two vectors
     * @param other Other vector
     * @return Distance between vectors
     */
    public double distance(Vector2 other) {
        double dx = this.x - other.x;
        double dy = this.y - other.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    /**
     * Calculates the squared distance between two vectors
     * @param other Other vector
     * @return Squared distance between vectors
     */
    public double distanceSquared(Vector2 other) {
        double dx = this.x - other.x;
        double dy = this.y - other.y;
        return dx * dx + dy * dy;
    }

    /**
     * Linear interpolation between two vectors
     * @param other Target vector
     * @param t Interpolation factor (0.0 = this, 1.0 = other)
     * @return Interpolated vector
     */
    public Vector2 lerp(Vector2 other, double t) {
        t = Math.max(0.0, Math.min(1.0, t)); // Clamp t to [0,1]
        return new Vector2(
            this.x + (other.x - this.x) * t,

```

```

        this.y + (other.y - this.y) * t
    );
}

// --- Utility Methods ---

/***
 * Returns the negated vector
 * @return New negated vector
 */
public Vector2 negate() {
    return new Vector2(-x, -y);
}

/***
 * Returns the absolute value of each component
 * @return New vector with absolute components
 */
public Vector2 abs() {
    return new Vector2(Math.abs(x), Math.abs(y));
}

/***
 * Clamps the vector components to specified range
 * @param min Minimum value
 * @param max Maximum value
 * @return New clamped vector
 */
public Vector2 clamp(double min, double max) {
    return new Vector2(
        Math.max(min, Math.min(max, x)),
        Math.max(min, Math.min(max, y))
    );
}

/***
 * Checks if vector is approximately zero
 * @param epsilon Tolerance value
 * @return True if both components are near zero
 */

```

```
 */
public boolean isZero(double epsilon) {
    return Math.abs(x) < epsilon && Math.abs(y) < epsilon;
}

/**
 * Checks if vector is exactly zero
 * @return True if both components are zero
 */
public boolean isZero() {
    return x == 0.0 && y == 0.0;
}

// --- Factory Methods ---

/**
 * Creates a zero vector
 * @return Zero vector
 */
public static Vector2 zero() {
    return new Vector2(0.0, 0.0);
}

/**
 * Creates a unit vector in X direction
 * @return Unit X vector
 */
public static Vector2 unitX() {
    return new Vector2(1.0, 0.0);
}

/**
 * Creates a unit vector in Y direction
 * @return Unit Y vector
 */
public static Vector2 unitY() {
    return new Vector2(0.0, 1.0);
}
```

```
/**  
 * Creates a vector with both components set to value  
 * @param value Component value  
 * @return New vector  
 */  
public static Vector2 fill(double value) {  
    return new Vector2(value, value);  
}  
  
// --- Object Overrides ---
```

@Override

```
public boolean equals(Object obj) {  
    if (this == obj) return true;  
    if (obj == null || getClass() != obj.getClass()) return false;  
    Vector2 other = (Vector2) obj;  
    return Double.compare(other.x, x) == 0 && Double.compare(other.y,  
y) == 0;  
}
```

@Override

```
public int hashCode() {  
    long xBits = Double.doubleToLongBits(x);  
    long yBits = Double.doubleToLongBits(y);  
    return (int)(xBits ^ (xBits >>> 32)) ^ (int)(yBits ^ (yBits >>> 32));  
}
```

@Override

```
public String toString() {  
    return String.format("Vector2(%f, %f)", x, y);  
}
```

```
// --- Additional Methods for Graphics ---
```

```
/**  
 * Rotates the vector by specified angle (in radians)  
 * @param angle Angle in radians  
 * @return New rotated vector  
 */
```

```

public Vector2 rotate(double angle) {
    double cos = Math.cos(angle);
    double sin = Math.sin(angle);
    return new Vector2(
        x * cos - y * sin,
        x * sin + y * cos
    );
}

/**
 * Returns the perpendicular vector (90 degree rotation)
 * @return New perpendicular vector
 */
public Vector2 perpendicular() {
    return new Vector2(-y, x);
}

/**
 * Returns the angle of the vector in radians
 * @return Angle in radians [-π, π]
 */
public double angle() {
    return Math.atan2(y, x);
}

/**
 * Returns the angle between two vectors in radians
 * @param other Other vector
 * @return Angle between vectors in radians [0, π]
 */
public double angleBetween(Vector2 other) {
    double dot = this.dot(other);
    double len1 = this.length();
    double len2 = other.length();

    if (len1 < 1e-10 || len2 < 1e-10) {
        return 0.0;
    }
}

```

```

        return Math.acos(dot / (len1 * len2));
    }

    /**
     * Projects this vector onto another vector
     * @param other Vector to project onto
     * @return Projection vector
     */
    public Vector2 project(Vector2 other) {
        double lenSq = other.lengthSquared();
        if (lenSq < 1e-10) {
            return Vector2.zero();
        }
        double scale = this.dot(other) / lenSq;
        return other.multiply(scale);
    }

    /**
     * Reflects this vector across a normal vector
     * @param normal Normal vector (should be unit length)
     * @return Reflected vector
     */
    public Vector2 reflect(Vector2 normal) {
        double dot = this.dot(normal);
        return this.subtract(normal.multiply(2.0 * dot));
    }

}

// =====
// File: /net/lena/murat/math/Matrix4.java
// =====

package net.elena.murat.math;

import net.elena.murat.math.Ray;

/**

```

```

* Represents a 4x4 matrix for 3D transformations (translation, rotation,
scaling).
*/
public class Matrix4 {
    private final double[][] m; // Matrix elements

    /**
     * Constructs an identity Matrix4.
     */
    public Matrix4() {
        m = new double[4][4];
        m[0][0] = 1.0; m[0][1] = 0.0; m[0][2] = 0.0; m[0][3] = 0.0;
        m[1][0] = 0.0; m[1][1] = 1.0; m[1][2] = 0.0; m[1][3] = 0.0;
        m[2][0] = 0.0; m[2][1] = 0.0; m[2][2] = 1.0; m[2][3] = 0.0;
        m[3][0] = 0.0; m[3][1] = 0.0; m[3][2] = 0.0; m[3][3] = 1.0;
    }

    /**
     * Constructs a Matrix4 with the specified elements.
     */
    public Matrix4(double m00, double m01, double m02, double m03,
                  double m10, double m11, double m12, double m13,
                  double m20, double m21, double m22, double m23,
                  double m30, double m31, double m32, double m33) {
        m = new double[4][4];
        this.m[0][0] = m00; this.m[0][1] = m01; this.m[0][2] = m02; this.m[0]
        [3] = m03;
        this.m[1][0] = m10; this.m[1][1] = m11; this.m[1][2] = m12; this.m[1]
        [3] = m13;
        this.m[2][0] = m20; this.m[2][1] = m21; this.m[2][2] = m22; this.m[2]
        [3] = m23;
        this.m[3][0] = m30; this.m[3][1] = m31; this.m[3][2] = m32; this.m[3]
        [3] = m33;
    }

    /**
     * Constructs a new Matrix4 by copying an existing matrix.
     * @param other The Matrix4 object to copy.
     */

```

```

public Matrix4(Matrix4 other) {
    this(other.m[0][0], other.m[0][1], other.m[0][2], other.m[0][3],
        other.m[1][0], other.m[1][1], other.m[1][2], other.m[1][3],
        other.m[2][0], other.m[2][1], other.m[2][2], other.m[2][3],
        other.m[3][0], other.m[3][1], other.m[3][2], other.m[3][3]);
}

/**
 * Returns an identity (unit) 4x4 matrix.
 * An identity matrix has 1s on the main diagonal and 0s elsewhere.
 * It represents no translation, rotation, or scaling.
 *
 * @return A new 4x4 identity matrix.
 */
public static Matrix4 identity() {
    return new Matrix4(); // The default constructor creates an identity
matrix
}

/**
 * Sets the value at the specified row and column.
 * @param row The row index (0-3)
 * @param col The column index (0-3)
 * @param value The value to set
 * @throws IndexOutOfBoundsException if row or col is not in [0, 3]
 */
public void set(int row, int col, double value) {
    if (row < 0 || row >= 4 || col < 0 || col >= 4) {
        throw new IndexOutOfBoundsException("Matrix4 indices out of
bounds: [" + row + "][" + col + "]");
    }

    this.m[row][col] = value;
}

/**
 * Gets the X-axis scale factor from this transformation matrix.
 * This is calculated as the magnitude of the X basis vector.
 * @return The X scale factor

```

```

*/
public double getScaleX() {
    return Math.sqrt(m[0][0] * m[0][0] + m[1][0] * m[1][0] + m[2][0] *
m[2][0]);
}

/***
 * Gets the Y-axis scale factor from this transformation matrix.
 * This is calculated as the magnitude of the Y basis vector.
 * @return The Y scale factor
*/
public double getScaleY() {
    return Math.sqrt(m[0][1] * m[0][1] + m[1][1] * m[1][1] + m[2][1] *
m[2][1]);
}

/***
 * Gets the Z-axis scale factor from this transformation matrix.
 * This is calculated as the magnitude of the Z basis vector.
 * @return The Z scale factor
*/
public double getScaleZ() {
    return Math.sqrt(m[0][2] * m[0][2] + m[1][2] * m[1][2] + m[2][2] *
m[2][2]);
}

public Ray transformRay(Ray ray) {
    Point3 newOrigin = this.transformPoint(ray.getOrigin());
    Vector3 newDirection =
this.transformVector(ray.getDirection()).normalize();
    return new Ray(newOrigin, newDirection);
}

/***
 * Transforms a direction vector by this matrix.
 * Unlike points, vectors are not affected by translation.
 * Only the rotational and scaling components are applied.
 *
 * This is used for transforming normal vectors, ray directions, etc.

```

```

*
* @param v The direction vector to transform
* @return A new transformed Vector3
*/
public Vector3 transformDirection(Vector3 v) {
    double x = m[0][0] * v.x + m[0][1] * v.y + m[0][2] * v.z;
    double y = m[1][0] * v.x + m[1][1] * v.y + m[1][2] * v.z;
    double z = m[2][0] * v.x + m[2][1] * v.y + m[2][2] * v.z;
    return new Vector3(x, y, z);
}

/**
* Provides access to a specific element of the matrix.
* @param row The row index (0-3).
* @param col The column index (0-3).
* @return The matrix element at the specified position.
* @throws IndexOutOfBoundsException If the row or column index is
invalid.
*/
public double get(int row, int col) {
    if (row < 0 || row >= 4 || col < 0 || col >= 4) {
        throw new IndexOutOfBoundsException("Matrix4 indices out of
bounds: [" + row + "][" + col + "]");
    }
    return m[row][col];
}

/**
* Normal vektörü dönüştürür (normal transformasyonu için).
* Normal vektörlerin doğru dönüşümü için matrisin ters transpozu
kullanılır.
* @param normal Dönüşürtülecek normal vektör
* @return Dönüşürtülmüş normal vektör (normalize edilmiş)
*/
public Vector3 transformNormal(Vector3 normal) {
    // Matrisin ters transpozu alınır
    Matrix4 normalMatrix = this.inverseTransposeForNormal();

    if (normalMatrix == null) {

```

```

        return new Vector3(0, 0, 0); // Geçersiz dönüşüm durumu
    }

    // Vektörü dönüştür (w=0 varsayılarak, sadece 3x3 kısım kullanılır)
    double x = normal.x;
    double y = normal.y;
    double z = normal.z;

    double newX = normalMatrix.m[0][0] * x + normalMatrix.m[0][1] * y
    + normalMatrix.m[0][2] * z;
    double newY = normalMatrix.m[1][0] * x + normalMatrix.m[1][1] * y
    + normalMatrix.m[1][2] * z;
    double newZ = normalMatrix.m[2][0] * x + normalMatrix.m[2][1] * y +
    normalMatrix.m[2][2] * z;

    return new Vector3(newX, newY, newZ).normalize();
}

/***
 * Multiplies this matrix by another matrix.
 * @param other The other Matrix4 to multiply with.
 * @return The resulting Matrix4.
 */
public Matrix4 multiply(Matrix4 other) {
    Matrix4 result = new Matrix4(); // Start with an identity matrix
    for (int i = 0; i < 4; i++) {
        for (int j = 0; j < 4; j++) {
            result.m[i][j] = 0; // Reset element before summing
            for (int k = 0; k < 4; k++) {
                result.m[i][j] += this.m[i][k] * other.m[k][j];
            }
        }
    }
    return result;
}

/***
 * Transforms a Point3 by this matrix (includes translation).
 * For affine transformations, the W component of the transformed point

```

should be 1.0.

```
* @param point The Point3 to transform.  
* @return The transformed Point3.  
*/  
public Point3 transformPoint(Point3 point) {  
    double x = m[0][0] * point.x + m[0][1] * point.y + m[0][2] * point.z +  
    m[0][3];  
    double y = m[1][0] * point.x + m[1][1] * point.y + m[1][2] * point.z +  
    m[1][3];  
    double z = m[2][0] * point.x + m[2][1] * point.y + m[2][2] * point.z +  
    m[2][3];  
    return new Point3(x, y, z);  
}  
  
/**  
 * Transforms a Vector3 by this matrix (only rotation and scaling, no  
translation).  
* @param vector The Vector3 to transform.  
* @return The transformed Vector3.  
*/  
public Vector3 transformVector(Vector3 vector) {  
    double x = m[0][0] * vector.x + m[0][1] * vector.y + m[0][2] * vector.z;  
    double y = m[1][0] * vector.x + m[1][1] * vector.y + m[1][2] * vector.z;  
    double z = m[2][0] * vector.x + m[2][1] * vector.y + m[2][2] * vector.z;  
    return new Vector3(x, y, z);  
}  
  
/**  
 * Returns the inverse of this matrix. Returns null if the matrix is non-  
invertible.  
* This method is designed for affine transformations (rotation,  
translation, uniform scaling).  
* Formula: [ R | t ]^-1 = [ R^-1 | -R^-1 * t ]  
* Where R is the upper-left 3x3 submatrix and t is the translation vector.  
* @return The inverse Matrix4 or null.  
*/  
public Matrix4 inverse() {  
    // Extract the upper 3x3 rotation/scale part  
    Matrix3 upperLeft = new Matrix3(
```

```

        m[0][0], m[0][1], m[0][2],
        m[1][0], m[1][1], m[1][2],
        m[2][0], m[2][1], m[2][2]
    );
    Matrix3 invUpperLeft = upperLeft.inverse(); // This performs its own
determinant check

    if (invUpperLeft == null) {
        System.err.println("Warning: Upper 3x3 part of Matrix4 is non-
invertible, cannot compute inverse.");
        return null;
    }

    Matrix4 inv = new Matrix4(); // Resulting inverse matrix, initialized to
identity

    // Set the upper-left 3x3 of the inverse matrix ( $R^{-1}$ )
    inv.m[0][0] = invUpperLeft.get(0,0); inv.m[0][1] =
invUpperLeft.get(0,1); inv.m[0][2] = invUpperLeft.get(0,2);
    inv.m[1][0] = invUpperLeft.get(1,0); inv.m[1][1] =
invUpperLeft.get(1,1); inv.m[1][2] = invUpperLeft.get(1,2);
    inv.m[2][0] = invUpperLeft.get(2,0); inv.m[2][1] =
invUpperLeft.get(2,1); inv.m[2][2] = invUpperLeft.get(2,2);

    // Calculate the inverse translation part:  $-R^{-1} * t$ 
    Vector3 translation = new Vector3(m[0][3], m[1][3], m[2][3]);
    Vector3 invTranslation = invUpperLeft.transform(translation).negate();

    inv.m[0][3] = invTranslation.x;
    inv.m[1][3] = invTranslation.y;
    inv.m[2][3] = invTranslation.z;

    // Bottom row remains [0, 0, 0, 1] for affine transformations
    inv.m[3][0] = 0.0; inv.m[3][1] = 0.0; inv.m[3][2] = 0.0; inv.m[3][3] =
1.0;

    return inv;
}

```

```

/***
 * Computes the inverse transpose of the upper 3x3 part of this matrix.
 * This is typically used to transform normal vectors correctly when the
 * model matrix contains non-uniform scaling.
 * For pure rotations, the inverse is equal to the transpose.
 *
 * @return A new Matrix4 representing the inverse transpose of the 3x3
part,
 * with the translation components set to zero. Returns null if the
 * upper 3x3 part is non-invertible.
 */
public Matrix4 inverseTransposeForNormal() {
    // Extract the upper 3x3 part
    Matrix3 upperLeft = new Matrix3(
        m[0][0], m[0][1], m[0][2],
        m[1][0], m[1][1], m[1][2],
        m[2][0], m[2][1], m[2][2]
    );

    // Compute its inverse
    Matrix3 invUpperLeft = upperLeft.inverse();

    if (invUpperLeft == null) {
        System.err.println("Warning: Upper 3x3 part of Matrix4 is non-
invertible, cannot compute inverse transpose for normal.");
        return null;
    }

    // Transpose the inverse (this is the correct operation for normals)
    Matrix3 normalMatrix3 = invUpperLeft.transpose();

    // Construct a new Matrix4 from this 3x3, with translation part zeroed
out
    return new Matrix4(
        normalMatrix3.get(0,0), normalMatrix3.get(0,1),
        normalMatrix3.get(0,2), 0,
        normalMatrix3.get(1,0), normalMatrix3.get(1,1),
        normalMatrix3.get(1,2), 0,
        normalMatrix3.get(2,0), normalMatrix3.get(2,1),
    );
}

```

```

normalMatrix3.get(2,2), 0,
    0, 0, 0, 1
);
}

/***
 * Creates a translation matrix.
 * @param translation The translation vector.
 * @return The translation Matrix4.
 */
public static Matrix4 translate(Vector3 translation) {
    return new Matrix4(
        1, 0, 0, translation.x,
        0, 1, 0, translation.y,
        0, 0, 1, translation.z,
        0, 0, 0, 1
    );
}

public static Matrix4 translate(double x, double y, double z) {
    return new Matrix4(
        1, 0, 0, x,
        0, 1, 0, y,
        0, 0, 1, z,
        0, 0, 0, 1
    );
}

/***
 * Creates a rotation matrix around the X-axis.
 * @param angleDegrees The rotation angle in degrees.
 * @return The rotation Matrix4.
 */
public static Matrix4 rotateX(double angleDegrees) {
    double angleRad = Math.toRadians(angleDegrees);
    double cosA = Math.cos(angleRad);
    double sinA = Math.sin(angleRad);
    return new Matrix4(
        1, 0, 0, 0,

```

```

        0, cosA, -sinA, 0,
        0, sinA, cosA, 0,
        0, 0, 0, 1
    );
}

/***
 * Creates a rotation matrix around the Y-axis.
 * @param angleDegrees The rotation angle in degrees.
 * @return The rotation Matrix4.
 */
public static Matrix4 rotateY(double angleDegrees) {
    double angleRad = Math.toRadians(angleDegrees);
    double cosA = Math.cos(angleRad);
    double sinA = Math.sin(angleRad);
    return new Matrix4(
        cosA, 0, sinA, 0,
        0, 1, 0, 0,
        -sinA, 0, cosA, 0,
        0, 0, 0, 1
    );
}

/***
 * Creates a rotation matrix around the Z-axis.
 * @param angleDegrees The rotation angle in degrees.
 * @return The rotation Matrix4.
 */
public static Matrix4 rotateZ(double angleDegrees) {
    double angleRad = Math.toRadians(angleDegrees);
    double cosA = Math.cos(angleRad);
    double sinA = Math.sin(angleRad);
    return new Matrix4(
        cosA, -sinA, 0, 0,
        sinA, cosA, 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
    );
}

```

```

/**
 * Creates a scaling matrix with the specified scale factors.
 * @param sx The X-axis scale factor.
 * @param sy The Y-axis scale factor.
 * @param sz The Z-axis scale factor.
 * @return The scaling Matrix4.
 */
public static Matrix4 scale(double sx, double sy, double sz) {
    return new Matrix4(
        sx, 0, 0, 0,
        0, sy, 0, 0,
        0, 0, sz, 0,
        0, 0, 0, 1
    );
}

// Matrix4 sınıfına bu metodu ekleyin
public Matrix4 transpose() {
    return new Matrix4(
        m[0][0], m[1][0], m[2][0], m[3][0],
        m[0][1], m[1][1], m[2][1], m[3][1],
        m[0][2], m[1][2], m[2][2], m[3][2],
        m[0][3], m[1][3], m[2][3], m[3][3]
    );
}

/**
 * Creates a Matrix4 from a Matrix3 (typically to extend rotation matrices
 * to 4x4).
 * @param m3 The Matrix3 to extend.
 * @return The created Matrix4.
 */
public static Matrix4 fromMatrix3(Matrix3 m3) {
    return new Matrix4(
        m3.get(0,0), m3.get(0,1), m3.get(0,2), 0,
        m3.get(1,0), m3.get(1,1), m3.get(1,2), 0,
        m3.get(2,0), m3.get(2,1), m3.get(2,2), 0,
        0, 0, 0, 1
    );
}

```

```
    );
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 4; i++) {
        sb.append("| ");
        for (int j = 0; j < 4; j++) {
            sb.append(String.format("%8.4f", m[i][j])).append(" ");
        }
        sb.append("\n");
    }
    return sb.toString();
}
```

```
// =====
// File: /net/elenamurat/math/Ray.java
// =====
```

```
package net.elena.murat.math;

/**
 * Represents a ray in 3D space, defined by an origin point and a direction
 * vector.
 * Includes energy tracking for ray tracing optimizations.
 */
public class Ray {
    private final Point3 origin;
    private final Vector3 direction;
    private double energy; // New field for energy tracking

    /**
     * A small constant used to offset ray origins to prevent self-intersection.
     */
    public static final double EPSILON = 1e-4;//original is 1e-4
```

```

/***
 * Constructs a new Ray with full energy (1.0).
 * The direction vector will be normalized automatically.
 * @param origin The origin point of the ray.
 * @param direction The direction vector of the ray.
 */
public Ray(Point3 origin, Vector3 direction) {
    this(origin, direction, 1.0); // Default full energy
}

/***
 * Constructs a new Ray with specified energy.
 * @param origin The origin point of the ray.
 * @param direction The direction vector of the ray (will be normalized).
 * @param energy Initial energy of the ray (0.0 to 1.0).
 */
public Ray(Point3 origin, Vector3 direction, double energy) {
    this.origin = origin;
    this.direction = direction.normalize();
    this.energy = Math.max(0, Math.min(1, energy)); // clamp
}

/***
 * Gets the current energy of the ray.
 * @return Energy value between 0.0 and 1.0.
 */
public double getEnergy() {
    return energy;
}

/***
 * Sets the energy of the ray.
 * @param energy New energy value (will be clamped to [0.0, 1.0]).
 */
public void setEnergy(double energy) {
    this.energy = Math.max(0.0, Math.min(1.0, energy));
}

```

```

/**
 * Creates a new ray with scaled energy (useful for
reflections/refractions).
 * @param energyFactor Factor to multiply current energy by (0.0 to
1.0).
 * @return New Ray instance with adjusted energy.
 */
public Ray createChildRay(double energyFactor) {
    return new Ray(
        this.origin,
        this.direction,
        this.energy * Math.max(0.0, Math.min(1.0, energyFactor))
    );
}

// Existing methods remain unchanged:
public Point3 getOrigin() {
    return origin;
}

public Vector3 getDirection() {
    return direction;
}

public Point3 pointAtParameter(double t) {
    return origin.add(direction.scale(t));
}

/**
 * Transforms this ray by the given transformation matrix.
 * The origin point is transformed as a point (w=1).
 * The direction vector is transformed as a vector (w=0).
 *
 * This is commonly used to convert a world-space ray into object space
 * by applying the object's inverse transformation matrix.
 *
 * @param matrix The transformation matrix (usually inverse of object's
transform)
 * @return A new Ray in the transformed space

```

```
*/  
public Ray transform(Matrix4 matrix) {  
    // Transform the origin (treated as a point)  
    Point3 newOrigin = matrix.transformPoint(getOrigin());  
  
    // Transform the direction (treated as a vector)  
    Vector3 newDirection = matrix.transformDirection(getDirection());  
  
    return new Ray(newOrigin, newDirection);  
}  
  
@Override  
public String toString() {  
    return String.format("Ray(origin=%s, direction=%s, energy=%.3f)",  
        origin, direction, energy);  
}  
}  
  
// ======  
// File: /net/elenamurat/math/Vector3.java  
// ======
```

```
package net.elena.murat.math;  
  
import java.util.Optional;  
import java.util.Random;  
  
public class Vector3 {  
    public static final Vector3 ZERO = new Vector3(0, 0, 0);  
    public static final Vector3 ONE = new Vector3(1, 1, 1);  
  
    public final double x, y, z;  
  
    public Vector3(double x, double y, double z) {  
        this.x = x;  
        this.y = y;  
        this.z = z;
```

```

}

/**
 * Creates a vector from point 'from' to point 'to'
 * @param from The starting point
 * @param to The ending point
 */
public Vector3(Point3 from, Point3 to) {
    this.x = to.x - from.x;
    this.y = to.y - from.y;
    this.z = to.z - from.z;
}

public double get(int axis) {
    switch(axis) {
        case 0: return x;
        case 1: return y;
        case 2: return z;
        default: throw new IllegalArgumentException("Axis must be 0, 1 or
2");
    }
}

// Vector3.java'ya bu metodları ekleyin
public Vector3 normalizeSafe() {
    double len = length();
    return len > 0 ? new Vector3(x/len, y/len, z/len) : new Vector3(0, 1, 0);
}

public double angleBetweenSafe(Vector3 other) {
    if (other == null) return Math.PI/2;
    double dot = this.dot(other);
    double magProduct = this.length() * other.length();
    return magProduct > 0 ? Math.acos(Math.max(-1, Math.min(1,
dot/magProduct))) : Math.PI/2;
}

public double angleBetween(Vector3 other) {
    double dot = this.dot(other);

```

```

        double magProduct = this.length() * other.length();
        return Math.acos(dot / magProduct);
    }

    public double length() {
        return Math.sqrt(x * x + y * y + z * z);
    }

    public double lengthSquared() {
        return x * x + y * y + z * z;
    }

    public Vector3 multiply(double t) {
        return new Vector3(x * t, y * t, z * t);
    }

    public Vector3 lerp(Vector3 other, double t) {
        t = Math.max(0.0, Math.min(1.0, t));
        return new Vector3(
            x + (other.x - x) * t,
            y + (other.y - y) * t,
            z + (other.z - z) * t
        );
    }

    public Vector3 normalize() {
        double len = length();
        return len > 0 ? new Vector3(x/len, y/len, z/len) : new Vector3(0,0,0);
    }

    public Vector3 add(Vector3 other) {
        return new Vector3(x + other.x, y + other.y, z + other.z);
    }

    public Vector3 subtract(Vector3 other) {
        return new Vector3(x - other.x, y - other.y, z - other.z);
    }

    public Vector3 subtract(Point3 other) {

```

```

        return new Vector3(x - other.x, y - other.y, z - other.z);
    }

public Vector3 scale(double scalar) {
    return new Vector3(x * scalar, y * scalar, z * scalar);
}

public double dot(Vector3 other) {
    return x * other.x + y * other.y + z * other.z;
}

public Vector3 negate() {
    return new Vector3(-x, -y, -z);
}

public Vector3 cross(Vector3 other) {
    return new Vector3(
        y * other.z - z * other.y,
        z * other.x - x * other.z,
        x * other.y - y * other.x
    );
}

public Vector3 reflect(Vector3 normal) {
    return this.subtract(normal.scale(2 * this.dot(normal)));
}

//Original
public Optional<Vector3> refract(Vector3 normal, double n1, double n2)
{
    double n = n1 / n2;
    //double cosI = -this.dot(normal); //Original
    double cosI = this.dot(normal);
    double sinT2 = n * n * (1.0 - (cosI * cosI));

    if (sinT2 > 1.0) return Optional.empty();
    double cosT = Math.sqrt(1.0 - sinT2);
    return Optional.of(this.scale(n).add(normal.scale((n * cosI) -
cosT)).normalize());
}

```

```
}
```

```
public Optional<Vector3> refract(Vector3 normal, double eta) {
    return this.refract(normal, 1.0, eta);
}

public Vector3 refractSimple(Vector3 normal, double eta) {
    double cosI = -this.dot(normal);
    double sinT2 = eta * eta * (1.0 - cosI * cosI);

    if (sinT2 > 1.0) {
        return this.reflect(normal); // Total internal reflection
    }

    double cosT = Math.sqrt(1.0 - sinT2);
    return this.scale(eta).add(normal.scale(eta * cosI - cosT)).normalize();
}

/***
 * Calculates Fresnel reflection coefficient
 * @param incident The incident ray direction (normalized)
 * @param normal The surface normal (normalized)
 * @param ior1 Index of refraction of first medium
 * @param ior2 Index of refraction of second medium
 * @return Fresnel reflection coefficient between 0 and 1
 */
public static double calculateFresnel(Vector3 viewDir, Vector3 normal,
    double ior1, double ior2) {
    double cosTheta = Math.abs(viewDir.dot(normal));
    cosTheta = Math.max(0.0, Math.min(1.0, cosTheta));

    double r0 = Math.pow((ior1 - ior2) / (ior1 + ior2), 2);
    double fresnel = r0 + (1.0 - r0) * Math.pow(1.0 - cosTheta, 5);

    return Math.max(0.0, Math.min(1.0, fresnel));
}

public static Vector3 randomInUnitSphere(Random rand) {
    Vector3 p;
```

```

do {
    p = new Vector3(
        rand.nextDouble() * 2 - 1,
        rand.nextDouble() * 2 - 1,
        rand.nextDouble() * 2 - 1
    );
} while (p.lengthSquared() >= 1.0);
return p;
}

/**
 * Verilen incident vektörünün, normal vektöre göre yansımaya vektörünü
 * hesaplar.
 *
 * @param incident Gelen vektör (ışın yönü).
 * @param normal Yansıma yüzeyinin normal vektörü.
 * @return Yansıma vektörü olarak yeni bir Vector3 nesnesi döner.
 */
public static Vector3 reflect(Vector3 incident, Vector3 normal) {
    // R = I - 2 * (I . N) * N
    // I: incident vektör
    // N: normal vektör
    return incident.subtract(normal.scale(2 * incident.dot(normal)));
}

public Vector3 transformNormal(Matrix4 matrix) {
    Matrix4 inverseTranspose = matrix.inverseTransposeForNormal();
    if (inverseTranspose == null) {
        return new Vector3(0, 0, 0);
    }
    return new Vector3(
        inverseTranspose.get(0, 0) * x + inverseTranspose.get(0, 1) * y +
        inverseTranspose.get(0, 2) * z,
        inverseTranspose.get(1, 0) * x + inverseTranspose.get(1, 1) * y +
        inverseTranspose.get(1, 2) * z,
        inverseTranspose.get(2, 0) * x + inverseTranspose.get(2, 1) * y +
        inverseTranspose.get(2, 2) * z
    ).normalize();
}

```

```
@Override
public String toString() {
    return String.format("(%.2f, %.2f, %.2f)", x, y, z);
}

}

// =====
// File: /net/elenamurat/math/FloatColor.java
// =====

package net.elena.murat.math;

import java.awt.Color;

/***
 * Represents a color with floating-point red, green, blue components.
 * Used for high-precision color calculations in ray tracing.
 * Values are typically in [0.0, 1.0] range.
 * Alpha channel is not supported - all operations are RGB-only.
 */
public class FloatColor {
    public final double r, g, b, a;

    /**
     * Constructs a FloatColor with red, green, and blue components.
     *
     * @param r Red component (0.0 - 1.0)
     * @param g Green component (0.0 - 1.0)
     * @param b Blue component (0.0 - 1.0)
     */
    public FloatColor(double r, double g, double b) {
        this.r = r;
        this.g = g;
        this.b = b;
        this.a = 1.0;
    }
}
```

```
public FloatColor(double r, double g, double b, double a) {
    this.r = r;
    this.g = g;
    this.b = b;
    this.a = a;
}

/**
 * Constructs a FloatColor from a java.awt.Color.
 * Alpha channel is ignored.
 *
 * @param color The AWT Color to convert
 */
public FloatColor(Color color) {
    this.r = color.getRed() / 255.0;
    this.g = color.getGreen() / 255.0;
    this.b = color.getBlue() / 255.0;
    this.a = color.getAlpha() / 255.0;
}

public double getR () {
    return this.r;
}

public double getG () {
    return this.g;
}

public double getB () {
    return this.b;
}

public double getA () {
    return this.a;
}

/**
 * Adds this color to another color component-wise.

```

```

*
* @param other The color to add
* @return A new FloatColor representing the sum
*/
public FloatColor add(FloatColor other) {
    return new FloatColor(
        this.r + other.r,
        this.g + other.g,
        this.b + other.b,
        Math.max(this.a, other.a)
    );
}

/**
 * Multiplies this color by a scalar factor.
 *
 * @param factor The multiplication factor
 * @return A new FloatColor with scaled components
 */
public FloatColor multiply(double factor) {
    return new FloatColor(this.r * factor, this.g * factor, this.b * factor,
this.a);
}

public FloatColor divide(double divisor) {
    if(divisor == 0) {
        return new FloatColor(0, 0, 0, this.a);
    }
    return new FloatColor(this.r / divisor, this.g / divisor, this.b / divisor,
this.a);
}

/**
 * Scales this color by a factor (same as multiply).
 *
 * @param factor The scale factor
 * @return A new FloatColor
 */
public FloatColor scale(double factor) {

```

```

        return this.multiply(factor);
    }

/***
 * Clamps all components (r, g, b) to the [0.0, 1.0] range.
 *
 * @return A new clamped FloatColor
 */
public FloatColor clamp01() {
    return new FloatColor(
        Math.max(0.0, Math.min(1.0, r)),
        Math.max(0.0, Math.min(1.0, g)),
        Math.max(0.0, Math.min(1.0, b)),
        Math.max(0.0, Math.min(1.0, a)))
};

}

/***
 * Checks if this color is nearly black.
 *
 * @param threshold The maximum value for each component to be
 * considered "black"
 * @return true if all components are below the threshold
 */
public boolean isBlack(double threshold) {
    return r < threshold && g < threshold && b < threshold;
}

/***
 * Converts this FloatColor to a java.awt.Color for rendering.
 * Components are clamped and scaled to 0-255 range.
 * Result is always fully opaque (alpha=255).
 *
 * @return A new Color object with RGB values
 */
public Color toAWTColor() {
    FloatColor clamped = clamp01();
    int r = (int) (clamped.r * 255.0);
    int g = (int) (clamped.g * 255.0);
    int b = (int) (clamped.b * 255.0);
}

```

```

//if (Math.random() < 0.0001) System.out.println("toAWTColor: r=" + r
+ ", g=" + g + ", b=" + b);

return new Color(r, g, b); // RGB constructor (fully opaque)
}

public int toARGB() {
    int aInt = (int)(Math.max(0, Math.min(1, a)) * 255);

    int rInt = (int)(Math.max(0, Math.min(1, r)) * 255);
    int gInt = (int)(Math.max(0, Math.min(1, g)) * 255);
    int bInt = (int)(Math.max(0, Math.min(1, b)) * 255);

    return (aInt << 24) | (rInt << 16) | (gInt << 8) | bInt;
}

/**
public int toARGB() {
    int rInt = (int) (Math.max(0, Math.min(1, r)) * 255);
    int gInt = (int) (Math.max(0, Math.min(1, g)) * 255);
    int bInt = (int) (Math.max(0, Math.min(1, b)) * 255);
    int aInt = (int) (Math.max(0, Math.min(1, a)) * 255);

    //if (Math.random () < 0.001) {
        //    System.out.println("toARGB - Input: (" + r + "," + g + "," + b + "," +
        a + ")");
        //    System.out.println("toARGB - Output: alpha=" + aInt + ", rgb=(" +
        rInt + "," + gInt + "," + bInt + ")");
    //}

    return (aInt << 24) | (rInt << 16) | (gInt << 8) | bInt;
}
*/
// Transparent pikseller için özel metod:
public int toARGBWithAlpha(int alpha) {
    int r = (int) (Math.max(0, Math.min(1, this.r)) * 255);
    int g = (int) (Math.max(0, Math.min(1, this.g)) * 255);

```

```

int b = (int) (Math.max(0, Math.min(1, this.b)) * 255);
int a = Math.max(0, Math.min(255, alpha));
return (a << 24) | (r << 16) | (g << 8) | b;
}

/***
 * Performs component-wise multiplication (Hadamard product) with
another color.
 *
 * @param other The other FloatColor
 * @return A new FloatColor with multiplied components
 */
public FloatColor multiply(FloatColor other) {
    return new FloatColor(
        this.r * other.r,
        this.g * other.g,
        this.b * other.b,
        this.a * other.a
    );
}

/***
 * Static helper: component-wise multiplication.
 */
public static FloatColor product(FloatColor a, FloatColor b) {
    return a.multiply(b);
}

/***
 * Static helper: multiply color by scalar.
 */
public static FloatColor product(FloatColor a, double b) {
    return new FloatColor(a.r * b, a.g * b, a.b * b);
}

/***
 * Checks if this color is smaller than another in all components.
 *
 * @param other The other color
 */

```

```

* @return true if r < other.r && g < other.g && b < other.b
*/
public boolean isSmaller(FloatColor other) {
    return r < other.r && g < other.g && b < other.b;
}

/***
* Returns the component-wise inverse (1/c).
* Prevents division by zero.
*
* @return A new FloatColor
*/
public FloatColor inverse() {
    return new FloatColor(
        r != 0.0 ? 1.0 / r : 0.0,
        g != 0.0 ? 1.0 / g : 0.0,
        b != 0.0 ? 1.0 / b : 0.0,
        this.a
    );
}

/***
* Subtracts another color (absorption) component-wise.
* Clamps result to 0.0 minimum.
*
* @param absorption The color to subtract
* @return A new FloatColor
*/
public FloatColor subtract(FloatColor absorption) {
    return new FloatColor(
        Math.max(0.0, this.r - absorption.r),
        Math.max(0.0, this.g - absorption.g),
        Math.max(0.0, this.b - absorption.b)
    );
}

/***
* Subtracts a constant value from all components.
*

```

```

* @param value The value to subtract
* @return A new FloatColor
*/
public FloatColor subtract(double value) {
    return new FloatColor(
        Math.max(0.0, this.r - value),
        Math.max(0.0, this.g - value),
        Math.max(0.0, this.b - value)
    );
}

// Predefined constants
public static final FloatColor BLACK = new FloatColor(0.0, 0.0, 0.0,
0.0);
public static final FloatColor WHITE = new FloatColor(1.0, 1.0, 1.0,
0.0);

}

// =====
// File: /net/elenamurat/math/Matrix3.java
// =====

```

```

package net.elena.murat.math;

import net.elena.murat.math.Ray;

/**
 * Represents a 3x3 matrix for linear transformations in 3D space.
 */
public class Matrix3 {
    private final double[][] m;

    /**
     * Constructs an identity Matrix3.
     */
    public Matrix3() {
        m = new double[3][3];
    }
}
```

```

m[0][0] = 1.0; m[0][1] = 0.0; m[0][2] = 0.0;
m[1][0] = 0.0; m[1][1] = 1.0; m[1][2] = 0.0;
m[2][0] = 0.0; m[2][1] = 0.0; m[2][2] = 1.0;
}

/***
 * Constructs a Matrix3 with the specified elements.
 */
public Matrix3(double m00, double m01, double m02,
    double m10, double m11, double m12,
    double m20, double m21, double m22) {
    m = new double[3][3];
    this.m[0][0] = m00; this.m[0][1] = m01; this.m[0][2] = m02;
    this.m[1][0] = m10; this.m[1][1] = m11; this.m[1][2] = m12;
    this.m[2][0] = m20; this.m[2][1] = m21; this.m[2][2] = m22;
}

/***
 * Constructs a new Matrix3 by copying an existing matrix.
 * @param other The Matrix3 object to copy.
 */
public Matrix3(Matrix3 other) {
    this(other.m[0][0], other.m[0][1], other.m[0][2],
        other.m[1][0], other.m[1][1], other.m[1][2],
        other.m[2][0], other.m[2][1], other.m[2][2]);
}

/***
 * Provides access to a specific element of the matrix.
 * @param row The row index (0-2).
 * @param col The column index (0-2).
 * @return The matrix element at the specified position.
 * @throws IndexOutOfBoundsException If the row or column index is
invalid.
 */
public double get(int row, int col) {
    if (row < 0 || row >= 3 || col < 0 || col >= 3) {
        throw new IndexOutOfBoundsException("Matrix3 indices out of
bounds: [" + row + "][" + col + "]");
    }
}

```

```

        }
        return m[row][col];
    }

/***
 * Multiplies this matrix by another matrix.
 * @param other The other Matrix3 to multiply with.
 * @return The resulting Matrix3.
 */
public Matrix3 multiply(Matrix3 other) {
    Matrix3 result = new Matrix3();
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result.m[i][j] = 0;
            for (int k = 0; k < 3; k++) {
                result.m[i][j] += this.m[i][k] * other.m[k][j];
            }
        }
    }
    return result;
}

/***
 * Transforms a Vector3 by this matrix.
 * @param vector The Vector3 to transform.
 * @return The transformed Vector3.
 */
public Vector3 transform(Vector3 vector) {
    double x = m[0][0] * vector.x + m[0][1] * vector.y + m[0][2] * vector.z;
    double y = m[1][0] * vector.x + m[1][1] * vector.y + m[1][2] * vector.z;
    double z = m[2][0] * vector.x + m[2][1] * vector.y + m[2][2] * vector.z;
    return new Vector3(x, y, z);
}

/***
 * Computes the determinant of this matrix.
 * @return The determinant value.
 */
public double determinant() {

```

```

        return m[0][0] * (m[1][1] * m[2][2] - m[1][2] * m[2][1]) -
        m[0][1] * (m[1][0] * m[2][2] - m[1][2] * m[2][0]) +
        m[0][2] * (m[1][0] * m[2][1] - m[1][1] * m[2][0]);
    }

/***
 * Computes the inverse of this matrix.
 * @return The inverse Matrix3, or null if the matrix is singular (non-invertible).
 */
public Matrix3 inverse() {
    double det = determinant();
    if (Math.abs(det) < Ray.EPSILON) { // Using Ray.EPSILON for floating point comparison
        // System.err.println("Warning: Matrix3 is singular, cannot compute inverse.");
        return null;
    }

    double invDet = 1.0 / det;

    Matrix3 inv = new Matrix3();
    inv.m[0][0] = (m[1][1] * m[2][2] - m[1][2] * m[2][1]) * invDet;
    inv.m[0][1] = (m[0][2] * m[2][1] - m[0][1] * m[2][2]) * invDet;
    inv.m[0][2] = (m[0][1] * m[1][2] - m[0][2] * m[1][1]) * invDet;

    inv.m[1][0] = (m[1][2] * m[2][0] - m[1][0] * m[2][2]) * invDet;
    inv.m[1][1] = (m[0][0] * m[2][2] - m[0][2] * m[2][0]) * invDet;
    inv.m[1][2] = (m[0][2] * m[1][0] - m[0][0] * m[1][2]) * invDet;

    inv.m[2][0] = (m[1][0] * m[2][1] - m[1][1] * m[2][0]) * invDet;
    inv.m[2][1] = (m[0][1] * m[2][0] - m[0][0] * m[2][1]) * invDet;
    inv.m[2][2] = (m[0][0] * m[1][1] - m[0][1] * m[1][0]) * invDet;

    return inv;
}

/***
 * Computes the transpose of this matrix.
 */

```

```

* @return The transposed Matrix3.
*/
public Matrix3 transpose() {
    Matrix3 result = new Matrix3();
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            result.m[i][j] = this.m[j][i];
        }
    }
    return result;
}

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < 3; i++) {
        sb.append("| ");
        for (int j = 0; j < 3; j++) {
            sb.append(String.format("%8.4f", m[i][j])).append(" ");
        }
        sb.append("\n");
    }
    return sb.toString();
}

}

// =====
// File: /net/elenamurat/math/Intersection.java
// =====

package net.elena.murat.math;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.shape.EMShape;

public class Intersection {

```

```
private final Point3 point; // Intersection point
private final Vector3 normal; // Normal vector at intersection point
private final double distance; // Distance from ray origin to intersection
point (t parameter)
private final EMShape shape; // Intersected object

public Intersection(Point3 point, Vector3 normal, double distance,
EMShape shape) {
    this.point = point;
    this.normal = normal;
    this.distance = distance;
    this.shape = shape;
}

public Point3 getPoint() {
    return point;
}

public Vector3 getNormal() {
    return normal;
}

public double getDistance() {
    return distance;
}

/**
 * Returns the distance of the intersection point from the ray origin ('t'
parameter).
 * Returns the same value as getDistance().
 * @return Intersection distance (t value).
 */
public double getT() {
    return distance;
}

public EMShape getShape() {
    return shape;
}
```

```
@Override
public String toString() {
    return "Intersection{" +
        "point=" + point +
        ", normal=" + normal +
        ", distance=" + distance +
        ", shape=" + (shape != null ? shape.getClass().getSimpleName() :
"null") +
        '}';
}
```

```
// =====
// File: /net/elenamurat/material/GradientTextMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.*;
import java.awt.geom.*;
import java.awt.image.*;
import java.util.Random;
import net.elena.murat.light.*;
import net.elena.murat.math.*;

public class GradientTextMaterial implements Material {
    private final Color bgStartColor;
    private final Color bgEndColor;
    private final Color textStartColor;
    private final Color textEndColor;
    private final String text;
    private final Font font;
    private final StripeDirection direction;
    private final double reflectivity;
    private final double ior;
    private final double transparency;
```

```
private Matrix4 objectInverseTransform;
private final int xOffset;
private final int yOffset;

private BufferedImage texture;
private Random random = new Random();

// Main constructor with all parameters
public GradientTextMaterial(Color bgStart, Color bgEnd,
    Color textStart, Color textEnd,
    String text, Font font, StripeDirection direction,
    double reflectivity, double ior, double transparency,
    Matrix4 objectInverseTransform,
    int xOffset, int yOffset) {
    this.bgStartColor = bgStart;
    this.bgEndColor = bgEnd;
    this.textStartColor = textStart;
    this.textEndColor = textEnd;
    this.text = text;
    this.font = font;
    this.direction = direction;
    this.reflectivity = Math.min(1.0, Math.max(0.0, reflectivity));
    this.ior = Math.max(1.0, ior);
    this.transparency = Math.min(1.0, Math.max(0.0, transparency));
    this.objectInverseTransform = objectInverseTransform;
    this.xOffset = xOffset;
    this.yOffset = yOffset;

    this.texture = createCompositeTexture();
}

// Simplified constructor with default parameters
public GradientTextMaterial(String text) {
    this(text, 0, 0);
}

// Constructor with text and position offsets
public GradientTextMaterial(String text, int xOffset, int yOffset) {
    this(
```

```
    generateRandomColor(),
    generateRandomColor(),
    Color.WHITE,
    Color.BLACK,
    text,
    new Font("Arial", Font.BOLD, 72),
    StripeDirection.RANDOM,
    0.3,
    1.0,
    0.1,
    new Matrix4 (),
    xOffset,
    yOffset
);
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}
```

```
private static Color generateRandomColor() {
    Random rand = new Random();
    return new Color(rand.nextFloat(), rand.nextFloat(), rand.nextFloat());
}
```

```
/**
 * Creates a texture with cyclic (repeating) gradients for strong visual
impact.
 * Uses GradientPaint with isCyclic=true to create wave-like color
transitions
 * that remain visible even after spherical mapping and on low-resolution
renders.
 *
 * @return A BufferedImage with repeating gradient patterns suitable for
3D materials.
 */
private BufferedImage createCompositeTexture() {
```

```

final int size = 512; // Lower resolution for better performance
BufferedImage texture = new BufferedImage(size, size,
BufferedImage.TYPE_INT_ARGB);
Graphics2D g2d = texture.createGraphics();

g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

// === 1. BACKGROUND: CYCLIC GRADIENT ===
StripeDirection bgDir = (direction == StripeDirection.RANDOM)
? StripeDirection.values()[random.nextInt(3)]
: direction;

Point2D bgStart = new Point2D.Float(0, 0);
Point2D bgEnd;

// Define a short vector to make gradient repeat frequently
switch (bgDir) {
    case HORIZONTAL:
        bgEnd = new Point2D.Float((float)(size * 0.2), 0.0f); // Repeat every
20% horizontally
        break;
    case VERTICAL:
        bgEnd = new Point2D.Float(0.0f, (float)(size * 0.2)); // Repeat every
20% vertically
        break;
    case DIAGONAL:
        bgEnd = new Point2D.Float((float)(size * 0.15), (float)(size *
0.15)); // Short diagonal
        break;
    default:
        bgEnd = new Point2D.Float((float)(size * 0.2), 0.0f);
}

// isCyclic = true → gradient repeats infinitely across the texture
GradientPaint bgGradient = new GradientPaint(
bgStart, bgStartColor,

```

```

bgEnd, bgEndColor,
true // <<< THIS LINE MAKES THE GRADIENT REPEATING
);
g2d.setPaint(bgGradient);
g2d.fillRect(0, 0, size, size);

// === 2. TEXT: CYCLIC GRADIENT MASK ===
if (text != null && !text.isEmpty()) {
    // Fix: Do not assign to 'font' if it's final
    // Use a local font or assume 'font' is already set
    Font renderFont = font != null ? font : new Font("Arial", Font.BOLD,
size / 6);
    g2d.setFont(renderFont);

    FontMetrics fm = g2d.getFontMetrics();
    int textWidth = fm.stringWidth(text);
    int textHeight = fm.getHeight();
    int ascent = fm.getAscent();

    int x = (size - textWidth) / 2 + xOffset;
    int y = (size - textHeight) / 2 + ascent + yOffset;

    x = Math.max(0, Math.min(size - textWidth, x));
    y = Math.max(ascent, Math.min(size - fm.getDescent(), y));

    // Define cyclic gradient direction for the text
    Point2D textStart = new Point2D.Float((float)x, (float)y);

    Point2D textEnd;

    switch (direction) {
        case HORIZONTAL:
            textEnd = new Point2D.Float(x + textWidth, y);
            break;
        case VERTICAL:
            textEnd = new Point2D.Float(x, y + textHeight);
            break;
        case DIAGONAL:
            textEnd = new Point2D.Float(x + textWidth, y + textHeight);
    }
}

```

```

        break;

    default:
        textEnd = new Point2D.Float(x + textWidth, y);
    }

    // Text gradient is also cyclic!
    GradientPaint textGradient = new GradientPaint(
        textStart, textStartColor,
        textEnd, textEndColor,
        true // <<< REPEATING TEXT GRADIENT
    );
    g2d.setPaint(textGradient);
    g2d.drawString(text, x, y);
}

g2d.dispose();
return texture;
}

private Point2D getEndPoint(int size, StripeDirection dir) {
    switch (dir) {
        case HORIZONTAL: return new Point2D.Float(size, 0);
        case VERTICAL: return new Point2D.Float(0, size);
        case DIAGONAL: return new Point2D.Float(size, size);
        default: return new Point2D.Float(size, 0);
    }
}

private Point2D getEndPoint(int width, int height, StripeDirection dir) {
    switch (dir) {
        case HORIZONTAL: return new Point2D.Float(width, 0);
        case VERTICAL: return new Point2D.Float(0, height);
        case DIAGONAL: return new Point2D.Float(width, height);
        default: return new Point2D.Float(width, 0);
    }
}

private Vector3 getLightDirection(Light light, Point3 worldPoint) {

```

```

if (light instanceof ElenaDirectionalLight) {
    return ((ElenaDirectionalLight)light).getDirection().normalize();
} else if (light instanceof MuratPointLight) {
    return
    ((MuratPointLight)light).getPosition().subtract(worldPoint).normalize();
} else if (light instanceof PulsatingPointLight) {
    return
    ((PulsatingPointLight)light).getPosition().subtract(worldPoint).normalize()
;
} else if (light instanceof BioluminescentLight) {
    return
    ((BioluminescentLight)light).getDirectionAt(worldPoint).normalize();
} else if (light instanceof BlackHoleLight) {
    return
    ((BlackHoleLight)light).getDirectionAt(worldPoint).normalize();
} else if (light instanceof FractalLight) {
    return ((FractalLight)light).getDirectionAt(worldPoint).normalize();
} else {
    System.err.println("Warning: Unsupported light type: " +
light.getClass().getName());
    return null;
}
}

```

```

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        return bgStartColor;
    }

    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    Color textureColor = getTextureColor(localPoint, localNormal);
    if (textureColor.getAlpha() == 0) {

```

```

        Color black = new Color(0, 0, 0, 1);
        return black;
    }
}
```

```

    return new Color(0, 0, 0, 0);
}

double texR = textureColor.getRed() / 255.0;
double texG = textureColor.getGreen() / 255.0;
double texB = textureColor.getBlue() / 255.0;

double rCombined = 0.0;
double gCombined = 0.0;
double bCombined = 0.0;

if (light instanceof ElenaMuratAmbientLight) {
    double ambientIntensity = light.getIntensity();
    rCombined = texR * ambientIntensity * (light.getColor().getRed() /
255.0);
    gCombined = texG * ambientIntensity * (light.getColor().getGreen() /
255.0);
    bCombined = texB * ambientIntensity * (light.getColor().getBlue() /
255.0);
} else {
    Vector3 lightDir = getLightDirection(light, worldPoint);
    if (lightDir != null) {
        double diffuseFactor = Math.max(0, worldNormal.dot(lightDir));
        rCombined = texR * diffuseFactor * (light.getColor().getRed() /
255.0) * light.getIntensity();
        gCombined = texG * diffuseFactor * (light.getColor().getGreen() /
255.0) * light.getIntensity();
        bCombined = texB * diffuseFactor * (light.getColor().getBlue() /
255.0) * light.getIntensity();

        if (diffuseFactor > 0) {
            Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
            Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
            double specFactor = Math.pow(Math.max(0,
viewDir.dot(reflectDir)), 32);

            rCombined += specFactor * (light.getColor().getRed() / 255.0);
            gCombined += specFactor * (light.getColor().getGreen() / 255.0);
            bCombined += specFactor * (light.getColor().getBlue() / 255.0);
        }
    }
}

```

```

        }
    }
}

return new Color(
    (float)Math.min(1.0, Math.max(0.0, rCombined)),
    (float)Math.min(1.0, Math.max(0.0, gCombined)),
    (float)Math.min(1.0, Math.max(0.0, bCombined)),
    textureColor.getAlpha() / 255.0f
);
}

/***
 * Maps a 3D point on the sphere to a 2D texture coordinate using
spherical mapping.
 * The texture is sampled with proper orientation, ensuring text appears
upright
 * when viewed from the front of the sphere.
 *
 * @param localPoint The point on the surface in object space.
 * @param localNormal The surface normal (unused here, kept for
interface).
 * @return The color sampled from the texture.
 */
private Color getTextureColor(Point3 localPoint, Vector3 localNormal) {
    if(texture == null) return bgStartColor;

    // Normalize direction vector from center to point
    Vector3 dir = new Vector3(localPoint.x, localPoint.y,
localPoint.z).normalize();

    // Convert to spherical coordinates
    double phi = Math.atan2(dir.z, dir.x);          // -π to π
    double theta = Math.asin(dir.y);                 // -π/2 to π/2

    // Map to UV [0,1]
    // U: Reverse the horizontal wrap so text appears correct
    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI); // Flip U horizontally
    double v = (theta + Math.PI / 2) / Math.PI;       // V: top to bottom
}

```

```

// Flip V because BufferedImage has Y-down
v = 1.0 - v;

// Wrap U for seamless tiling
int texX = (int)(u * texture.getWidth());
texX = texX % texture.getWidth();
if (texX < 0) texX += texture.getWidth();

// Clamp V
int texY = (int)(v * texture.getHeight());
if (texY < 0 || texY >= texture.getHeight()) {
    return new Color(0, 0, 0, 0);
}

return new Color(texture.getRGB(texX, texY), true);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

/**
EMShape sphere = new Sphere(1.2);

sphere.setTransform(Matrix4.translate(0, 1.2, 0));

```

```

Material material = new GradientTextMaterial(
Color.GREEN, Color.WHITE.darker (), //BG Colors
Color.RED, Color.BLUE,           // Gradient colors
"Takk",                      // Norwegian text
new Font("Arial", Font.BOLD, 200),// Font
GradientTextMaterial.StripeDirection.DIAGONAL, // Gradient direction
0.2, 1.0, 0.0,                // reflectivity, IOR, transparency
sphere.getInverseTransform(),   // object transform
0, 0 //x and y offset
);

sphere.setMaterial(material);
*/

```

```

// =====
// File: /net/elenamurat/material/CoffeeFjordMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CoffeeFjordMaterial implements Material {
    private final Color coffeeColor;
    private final Color fjordColor;
    private final double blendIntensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.45;
    private final double diffuseCoeff = 0.8;
    private final double specularCoeff = 0.25;
    private final double shininess = 35.0;
    private final double reflectivity = 0.15;
}

```

```

private final double ior = 2.1;
private final double transparency = 0.0;

public CoffeeFjordMaterial() {
    this(new Color(0x4A, 0x2C, 0x1D), new Color(0x1E, 0x90, 0xFF),
0.4);
}

public CoffeeFjordMaterial(Color coffeeColor, Color fjordColor, double
blendIntensity) {
    this.coffeeColor = coffeeColor;
    this.fjordColor = fjordColor;
    this.blendIntensity = Math.max(0, Math.min(1, blendIntensity));
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateMarbleEffect(objectPoint);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }
}

```

```

    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(new Color(0xFF, 0xF5,
0xE1), props.color, specularCoeff * specFactor);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateMarbleEffect(Point3 point) {
    double x = point.x * 6.0;
    double y = point.y * 6.0;
    double z = point.z * 6.0;

    // Marble-like turbulence effect
    double turbulence = Math.sin(x * 2.0 + Math.sin(y * 3.0) + Math.sin(z *
1.5 + Math.cos(x * 2.5)));
    double pattern = (turbulence + 2.0) / 4.0; // Normalize to 0-1

    if (pattern < blendIntensity) {
        double ratio = pattern / blendIntensity;
        return ColorUtil.blendColors(coffeeColor, fjordColor, ratio);
    } else {
        double ratio = (pattern - blendIntensity) / (1.0 - blendIntensity);
        Color darkCoffee = new Color(
            Math.max(0, coffeeColor.getRed() - 30),
            Math.max(0, coffeeColor.getGreen() - 20),
            Math.max(0, coffeeColor.getBlue() - 10)
        );
        return ColorUtil.blendColors(fjordColor, darkCoffee, ratio);
    }
}

```

```
@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/elenamurat/material/PolkaDotMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

public class PolkaDotMaterial implements Material {
    private Color baseColor;
    private Color dotColor;
    private double dotSize;
    private double dotSpacing;
    private double transparency;

    public PolkaDotMaterial(Color baseColor, Color dotColor,
```

```
double dotSize, double dotSpacing) {
    this.baseColor = baseColor;
    this.dotColor = dotColor;
    this.dotSize = dotSize;
    this.dotSpacing = dotSpacing;
    this.transparency = calculateTransparency(baseColor);
}

public PolkaDotMaterial(Color baseColor, Color dotColor) {
    this(baseColor, dotColor, 0.2, 0.5);
}

private double calculateTransparency(Color color) {
    int alpha = color.getAlpha();
    return 1.0 - ((double)alpha / 255.0);
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return 0.15;
}

@Override
public double getIndexOfRefraction() {
    return 1.0;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    // Not needed for this material
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
```

```

Point3 viewerPoint) {
    // Calculate pattern based on point coordinates
    double pattern = calculatePolkaDotPattern(point);

    // Use dot pattern as blend factor
    return ColorUtil.blendColors(baseColor, dotColor, pattern);
}

private double calculatePolkaDotPattern(Point3 point) {
    // Map 3D point to 2D plane for pattern generation
    double u = (point.x + 1000) % dotSpacing / dotSpacing;
    double v = (point.y + 1000) % dotSpacing / dotSpacing;
    double w = (point.z + 1000) % dotSpacing / dotSpacing;

    // Use different planes for more interesting pattern
    double distance = Math.sqrt(u * u + v * v);

    // Create circular dots
    if (distance < dotSize) {
        return 1.0; // Full dot color
    }

    // Add some variation using z-coordinate
    double distance2 = Math.sqrt(v * v + w * w);
    if (distance2 < dotSize * 0.8) {
        return 0.7; // Lighter dot color
    }

    return 0.0; // Base color
}

public Color getBaseColor() {
    return baseColor;
}

public void setBaseColor(Color baseColor) {
    this.baseColor = baseColor;
    this.transparency = calculateTransparency(baseColor);
}

```

```
public Color getDotColor() {
    return dotColor;
}

public void setDotColor(Color dotColor) {
    this.dotColor = dotColor;
}

public double getDotSize() {
    return dotSize;
}

public void setDotSize(double dotSize) {
    this.dotSize = dotSize;
}

public double getDotSpacing() {
    return dotSpacing;
}

public void setDotSpacing(double dotSpacing) {
    this.dotSpacing = dotSpacing;
}
```

```
// =====
// File: /net/elenamurat/material/NorwegianRoseMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;
```

```

public class NorwegianRoseMaterial implements Material {
    private final Color woodColor;
    private final Color roseColor;
    private Matrix4 objectTransform;
    private Matrix4 inverseTransform;
    private Matrix4 inverseTransposeTransform;

    // Phong parameters optimized for painted wood surface
    private final double ambientCoeff = 0.4;
    private final double diffuseCoeff = 0.7;
    private final double specularCoeff = 0.2;
    private final double shininess = 30.0;
    private final Color specularColor = new Color(220, 220, 220);
    private final double reflectivity = 0.1;
    private final double ior = 1.4;
    private final double transparency = 0.0;

    private static final double TWO_PI = Math.PI * 2;
    private static final double THREE_PI = Math.PI * 3;
    private static final double EIGHT = 8.0;

    public NorwegianRoseMaterial() {
        this(new Color(101, 67, 33), new Color(200, 50, 50));
    }

    public NorwegianRoseMaterial(Color woodColor, Color roseColor) {
        this.woodColor = woodColor;
        this.roseColor = roseColor;
        this.objectTransform = Matrix4.identity();
        this.inverseTransform = Matrix4.identity();
        this.inverseTransposeTransform = Matrix4.identity();
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectTransform = tm;
        this.inverseTransform = tm.inverse();
        this.inverseTransposeTransform = tm.inverseTransposeForNormal();
    }
}

```

```

}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // 1. Get rosemal pattern color with wood grain
    Color surfaceColor = calculateRosemalColor(worldPoint,
worldNormal);

    // 2. Handle light properties
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    // 3. Calculate Phong components
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateRosemalColor(Point3 worldPoint, Vector3
normal) {
    // Transform to object space for pattern calculation
}

```

```

Point3 localPoint = inverseTransform.transformPoint(worldPoint);
Vector3 localNormal =
inverseTransposeTransform.transformVector(normal).normalize();

// Calculate UV coordinates based on dominant normal axis
double u, v;
double absNx = Math.abs(localNormal.x);
double absNy = Math.abs(localNormal.y);
double absNz = Math.abs(localNormal.z);

if (absNx > absNy && absNx > absNz) {
    u = localPoint.y * EIGHT;
    v = localPoint.z * EIGHT;
} else if (absNy > absNx && absNy > absNz) {
    u = localPoint.x * EIGHT;
    v = localPoint.z * EIGHT;
} else {
    u = localPoint.x * EIGHT;
    v = localPoint.y * EIGHT;
}

// Add wood grain texture to base
Color woodBase = addWoodGrain(woodColor, u, v);

// Apply rosemål pattern on top
return createRosemalPattern(woodBase, u, v);
}

private Color addWoodGrain(Color baseWood, double u, double v) {
    // Simulate wood grain effect
    double grain = Math.sin(u * 3.0) * 0.15 + Math.sin(v * 1.5) * 0.1;

    int r = (int)(baseWood.getRed() * (0.85 + grain));
    int g = (int)(baseWood.getGreen() * (0.85 + grain));
    int b = (int)(baseWood.getBlue() * (0.85 + grain));

    r = r > 255 ? 255 : (r < 0 ? 0 : r);
    g = g > 255 ? 255 : (g < 0 ? 0 : g);
    b = b > 255 ? 255 : (b < 0 ? 0 : b);
}

```

```

    return new Color(r, g, b);
}

private Color createRosemalPattern(Color woodBase, double u, double v)
{
    double tileU = (u % 1 + 1) % 1;
    double tileV = (v % 1 + 1) % 1;

    // Border pattern
    if (tileU < 0.1 || tileU > 0.9 || tileV < 0.1 || tileV > 0.9) {
        return roseColor;
    }

    // Central flower/medallion pattern
    double centerX = 0.5;
    double centerY = 0.5;
    double dx = tileU - centerX;
    double dy = tileV - centerY;
    double distToCenterSq = dx * dx + dy * dy;

    if (distToCenterSq < 0.04) { // 0.2^2 = 0.04
        // Flower pattern with petals
        double angle = Math.atan2(dy, dx);
        double petal = Math.sin(angle * 6) * 0.5 + 0.5;
        if (petal > 0.7 && distToCenterSq > 0.01) { // 0.1^2 = 0.01
            return roseColor;
        }
    }

    // Scrollwork patterns
    double diffUV = Math.abs(tileU - tileV);
    double sumUV = Math.abs(tileU + tileV - 1);

    if (diffUV < 0.03 ||
        sumUV < 0.03 ||
        Math.sin(tileU * THREE_PI) > 0.8 ||
        Math.sin(tileV * THREE_PI) > 0.8 ||
        Math.cos(tileU * TWO_PI) > 0.7 ||

```

```

        Math.cos(tileV * TWO_PI) > 0.7) {
            return roseColor;
        }

        // Additional decorative elements
        double tileU25 = Math.abs(tileU - 0.25);
        double tileU75 = Math.abs(tileU - 0.75);
        double tileV25 = Math.abs(tileV - 0.25);
        double tileV75 = Math.abs(tileV - 0.75);

        if ((tileU25 < 0.04 && tileV25 < 0.04) ||
            (tileU75 < 0.04 && tileV75 < 0.04) ||
            (tileU25 < 0.04 && tileV75 < 0.04) ||
            (tileU75 < 0.04 && tileV25 < 0.04)) {
            return roseColor;
        }

        return woodBase;
    }

    @Override public double getReflectivity() { return reflectivity; }
    @Override public double getIndexOfRefraction() { return ior; }
    @Override public double getTransparency() { return transparency; }
}

```

```

// =====
// File: /net/elenamurat/material/FjordCrystalMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

```

```
public class FjordCrystalMaterial implements Material {  
    private final Color waterColor;  
    private final Color crystalColor;  
    private final double clarity;  
    private Matrix4 objectTransform;  
  
    private final double ambientCoeff = 0.3;  
    private final double diffuseCoeff = 0.6;  
    private final double specularCoeff = 0.8;  
    private final double shininess = 120.0;  
    private final double reflectivity = 0.7;  
    private final double ior = 1.33;  
    private final double transparency = 0.4;  
  
    public FjordCrystalMaterial() {  
        this(new Color(0x00, 0x7F, 0xFF), new Color(0xAF, 0xEE, 0xEE),  
        0.75);  
    }  
}
```

```
public FjordCrystalMaterial(Color waterColor, Color crystalColor,  
double clarity) {  
    this.waterColor = waterColor;  
    this.crystalColor = crystalColor;  
    this.clarity = Math.max(0, Math.min(1, clarity));  
    this.objectTransform = Matrix4.identity();  
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4();  
    this.objectTransform = tm;  
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light  
light, Point3 viewerPos) {  
    Point3 objectPoint =  
    objectTransform.inverse().transformPoint(worldPoint);
```

```

Color surfaceColor = calculateWaterEffect(objectPoint, worldNormal,
viewerPos);

LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
if (props == null) return surfaceColor;

Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(Color.WHITE, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateWaterEffect(Point3 point, Vector3 normal, Point3
viewerPos) {
    double x = point.x * 10.0;
    double y = point.y * 10.0;
    double z = point.z * 10.0;

    // Water caustics and wave patterns
    double wave1 = Math.sin(x * 1.5 + Math.sin(y * 0.8) * 2.0);
    double wave2 = Math.cos(y * 1.2 + Math.sin(z * 1.0) * 1.5);
    double wave3 = Math.sin(x * 2.0 + y * 1.7 + z * 0.5);
}

```

```

double waterPattern = (wave1 * 0.4 + wave2 * 0.3 + wave3 * 0.3);
double normalizedPattern = (waterPattern + 1.0) * 0.5;

// View-dependent transparency effect
Vector3 viewDir = viewerPos.subtract(point).normalize();
double viewAngle = Math.abs(viewDir.dot(normal));
double depthEffect = Math.pow(viewAngle, 0.5);

if (normalizedPattern < clarity) {
    // Crystal clear water areas
    double intensity = normalizedPattern / clarity * depthEffect;
    return ColorUtil.blendColors(waterColor, crystalColor, intensity);
} else {
    // Deeper water with more color saturation
    double intensity = (normalizedPattern - clarity) / (1.0 - clarity);
    Color deepWater = ColorUtil.darkerColor(waterColor, intensity * 0.4);
    return ColorUtil.addColorVariation(deepWater, intensity);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

```

```
// =====
// File: /net/elenamurat/material/MultiMixMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

public class MultiMixMaterial implements Material {

    private List<Material> materials;
    private List<Double> mixRatios;
    private Matrix4 objectTransform;
    private Matrix4 inverseObjectTransform;

    public MultiMixMaterial() {
        this.materials = new ArrayList<>();
        this.mixRatios = new ArrayList<>();
        this.objectTransform = Matrix4.identity();
        this.inverseObjectTransform = Matrix4.identity();
    }

    public MultiMixMaterial(Material[] materials, double[] ratios) {
        this();
        if (materials.length != ratios.length) {
            throw new IllegalArgumentException("Materials and ratios arrays must have same length");
        }

        for (int i = 0; i < materials.length; i++) {
            addMaterial(materials[i], ratios[i]);
        }
    }

    private void addMaterial(Material material, double ratio) {
        materials.add(material);
        mixRatios.add(ratio);
    }

    @Override
    public Color getDiffuseColor(Light light) {
        return null;
    }

    @Override
    public Matrix4 getObjectTransform() {
        return objectTransform;
    }

    @Override
    public Matrix4 getInverseObjectTransform() {
        return inverseObjectTransform;
    }

    @Override
    public String toString() {
        return "MultiMixMaterial{" +
                "materials=" + materials +
                ", mixRatios=" + mixRatios +
                '}';
    }
}
```

```

public void addMaterial(Material material, double ratio) {
    materials.add(material);
    mixRatios.add(Math.max(0.0, ratio));
    material.setObjectTransform(objectTransform);
}

public void removeMaterial(int index) {
    if(index >= 0 && index < materials.size()) {
        materials.remove(index);
        mixRatios.remove(index);
    }
}

public void setMaterialRatio(int index, double ratio) {
    if(index >= 0 && index < mixRatios.size()) {
        mixRatios.set(index, Math.max(0.0, ratio));
    }
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    if(materials.isEmpty()) return Color.BLACK;

    double totalRatio =
mixRatios.stream().mapToDouble(Double::doubleValue).sum();
    if(totalRatio == 0) return Color.BLACK;

    double totalRed = 0;
    double totalGreen = 0;
    double totalBlue = 0;

    for (int i = 0; i < materials.size(); i++) {
        Color c = materials.get(i).getColorAt(point, normal, light,
viewerPoint);
        double weight = mixRatios.get(i) / totalRatio;

        // Apply gamma correction for perceptual linear mixing
        double r = Math.pow(c.getRed() / 255.0, 2.2);

```

```

        double g = Math.pow(c.getGreen() / 255.0, 2.2);
        double b = Math.pow(c.getBlue() / 255.0, 2.2);

        totalRed += r * weight;
        totalGreen += g * weight;
        totalBlue += b * weight;
    }

    // Apply inverse gamma correction
    totalRed = Math.pow(totalRed, 1/2.2);
    totalGreen = Math.pow(totalGreen, 1/2.2);
    totalBlue = Math.pow(totalBlue, 1/2.2);

    int r = (int) Math.min(255, Math.max(0, totalRed * 255));
    int g = (int) Math.min(255, Math.max(0, totalGreen * 255));
    int b = (int) Math.min(255, Math.max(0, totalBlue * 255));

    return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    if (materials.isEmpty()) return 0.0;

    double totalRatio = 0.0;
    double weightedSum = 0.0;

    for (int i = 0; i < materials.size(); i++) {
        double ratio = mixRatios.get(i);
        weightedSum += materials.get(i).getReflectivity() * ratio;
        totalRatio += ratio;
    }

    return totalRatio > 0 ? weightedSum / totalRatio : 0.0;
}

```

```

@Override
public double getTransparency() {
    if (materials.isEmpty()) return 0.0;
}

```

```
double totalRatio = 0.0;
double weightedSum = 0.0;

for (int i = 0; i < materials.size(); i++) {
    double ratio = mixRatios.get(i);
    weightedSum += materials.get(i).getTransparency() * ratio;
    totalRatio += ratio;
}

return totalRatio > 0 ? weightedSum / totalRatio : 0.0;
}
```

```
@Override
public double getIndexOfRefraction() {
    if (materials.isEmpty()) return 1.0;

    double totalRatio = 0.0;
    double weightedSum = 0.0;

    for (int i = 0; i < materials.size(); i++) {
        double ratio = mixRatios.get(i);
        weightedSum += materials.get(i).getIndexOfRefraction() * ratio;
        totalRatio += ratio;
    }

    return totalRatio > 0 ? weightedSum / totalRatio : 1.0;
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    this.objectTransform = tm;
    this.inverseObjectTransform = tm.inverse();

    for (Material material : materials) {
        material.setObjectTransform(tm);
    }
}
```

```
private Color interpolateColors(Color color1, Color color2, double ratio)
{
    ratio = Math.max(0.0, Math.min(1.0, ratio));
    float[] comp1 = color1.getRGBColorComponents(null);
    float[] comp2 = color2.getRGBColorComponents(null);

    return new Color(
        (float) (comp1[0] * (1 - ratio) + comp2[0] * ratio),
        (float) (comp1[1] * (1 - ratio) + comp2[1] * ratio),
        (float) (comp1[2] * (1 - ratio) + comp2[2] * ratio)
    );
}

public int getMaterialCount() {
    return materials.size();
}

public Material getMaterial(int index) {
    return materials.get(index);
}

public double getRatio(int index) {
    return mixRatios.get(index);
}

public void normalizeRatios() {
    double total =
mixRatios.stream().mapToDouble(Double::doubleValue).sum();
    if (total > 0) {
        for (int i = 0; i < mixRatios.size(); i++) {
            mixRatios.set(i, mixRatios.get(i) / total);
        }
    }
}

public MultiMixMaterial withMaterial(Material material, double ratio) {
    addMaterial(material, ratio);
    return this;
}
```

```
}
```

```
// =====  
// File: /net/elenamurat/material/WordMaterial.java  
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.AlphaComposite;  
import java.awt.Color;  
import java.awt.Font;  
import java.awt.FontMetrics;  
import java.awt.GradientPaint;  
import java.awt.Graphics2D;  
import java.awt.RenderingHints;  
import java.awt.image.BufferedImage;
```

```
import net.elena.murat.light.Light;  
import net.elena.murat.math.Matrix4;  
import net.elena.murat.math.Point3;  
import net.elena.murat.math.Vector3;
```

```
/**
```

```
 * Material class that generates textures with rendered text and optional  
image on the fly.
```

```
 * Supports custom text, fonts, colors, gradients, transparent backgrounds,  
and image integration.
```

```
 * Uses planar UV mapping on XY plane (Z ignored) similar to  
TransparentPNGMaterial.
```

```
 */
```

```
public class WordMaterial implements Material {
```

```
private BufferedImage texture;  
private Matrix4 objectInverseTransform = new Matrix4();  
private double transparency = 1.0;
```

```
// UV parameters
```

```
private double uOffset = 0.0;
private double vOffset = 0.0;
private double uScale = 1.0;
private double vScale = 1.0;
private boolean isRepeatTexture = false;

private boolean isTriangleEtc = false;

// Text rendering parameters
private String text;
private Color foregroundColor;
private Color backgroundColor;
private Font font;
private boolean gradientEnabled;
private Color gradientColor;
private BufferedImage wordImage;
private int width;
private int height;

/**
 * Constructor with default styling (white text on transparent background,
 * Arial Bold 48)
 * @param text The text to render on the material
 */
public WordMaterial(String text) {
    this(text, Color.WHITE, new Color(0x00000000, true), new
Font("Arial", Font.BOLD, 48),
    false, null, null, 256, 256);
}

/**
 * Constructor with custom colors and font
 * @param text The text to render
 * @param foregroundColor Text color (RGB or ARGB)
 * @param backgroundColor Background color (use 0x00000000 for
transparent)
 * @param font The font to use for rendering
 */
public WordMaterial(String text, Color foregroundColor, Color
```

```
backgroundColor, Font font) {
    this(text, foregroundColor, backgroundColor, font, false, null, null, 256,
256);
}

/***
 * Constructor with gradient support
 * @param text The text to render
 * @param foregroundColor Starting gradient color
 * @param backgroundColor Background color
 * @param font The font to use
 * @param gradientColor Ending gradient color (if null, no gradient is
applied)
*/
public WordMaterial(String text, Color foregroundColor, Color
backgroundColor,
Font font, Color gradientColor) {
    this(text, foregroundColor, backgroundColor, font, true, gradientColor,
null, 256, 256);
}

/***
 * Constructor with image support
 * @param text The text to render
 * @param foregroundColor Text color
 * @param backgroundColor Background color
 * @param font The font to use
 * @param wordImage Optional image to display above text (null for text
only)
*/
public WordMaterial(String text, Color foregroundColor, Color
backgroundColor,
Font font, BufferedImage wordImage) {
    this(text, foregroundColor, backgroundColor, font, false, null,
wordImage,
    wordImage != null ? 384 : 256, wordImage != null ? 384 : 256);
}

/***
```

```
* Constructor with custom size
* @param text The text to render
* @param foregroundColor Text color
* @param backgroundColor Background color
* @param font The font to use
* @param width Texture width
* @param height Texture height
*/
public WordMaterial(String text, Color foregroundColor, Color
backgroundColor,
Font font, int width, int height) {
    this(text, foregroundColor, backgroundColor, font, false, null, null,
width, height);
}

/**
 * Full constructor with all parameters
* @param text The text to render
* @param foregroundColor Text color
* @param backgroundColor Background color
* @param font The font to use
* @param useGradient Whether to apply gradient effect
* @param gradientColor Gradient end color (required if useGradient is
true)
* @param wordImage Optional image to display above text (null for text
only)
* @param width Texture width
* @param height Texture height
*/
public WordMaterial(String text, Color foregroundColor, Color
backgroundColor,
Font font, boolean useGradient, Color gradientColor, BufferedImage
wordImage,
int width, int height) {
    this.text = text;
    this.foregroundColor = foregroundColor;
    this.backgroundColor = backgroundColor;
    this.font = font;
    this.gradientEnabled = useGradient;
```

```

this.gradientColor = gradientColor;
this.wordImage = wordImage;
this.width = width;
this.height = height;

    this.texture = createTextImage(text, foregroundColor, backgroundColor,
font,
    useGradient, gradientColor, wordImage, width, height);
}

/***
 * Creates a BufferedImage with the rendered text and optional image
 * @param text Text to render
 * @param fgColor Text color
 * @param bgColor Background color
 * @param font Font to use
 * @param useGradient Whether to use gradient
 * @param gradientColor Gradient end color
 * @param wordImage Optional image to display above text
 * @param width Image width
 * @param height Image height
 * @return BufferedImage with rendered content
*/
private static BufferedImage createTextImage(String text, Color fgColor,
Color bgColor,
Font font, boolean useGradient, Color gradientColor,
BufferedImage wordImage, int width, int height) {

    BufferedImage image = new BufferedImage(width, height,
BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = image.createGraphics();

    // Clear background (transparent or colored)
    if(bgColor.getAlpha() == 0) {
        g2d.setComposite(AlphaComposite.Clear);
        g2d.fillRect(0, 0, width, height);
        g2d.setComposite(AlphaComposite.SrcOver);
    } else {
        g2d.setColor(bgColor);

```

```

        g2d.fillRect(0, 0, width, height);
    }

    // Enable anti-aliasing for smooth rendering
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
RenderingHints.VALUE_RENDER_QUALITY);
    g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BICUBIC);

    // Draw word image if provided
    if(wordImage != null) {
        int imageSize = Math.min(width, height) - 64;
        int imageX = (width - imageSize) / 2;
        int imageY = 32;

        g2d.drawImage(wordImage, imageX, imageY, imageSize, imageSize,
null);
    }

    // Apply gradient or solid color for text
    if(useGradient && gradientColor != null) {
        int textY = (wordImage != null) ? (height * 2 / 3) : (height / 2);
        GradientPaint gradient = new GradientPaint(0, textY, fgColor, width,
textY + 50, gradientColor);
        g2d.setPaint(gradient);
    } else {
        g2d.setColor(fgColor);
    }

    // Set font - NO AUTO-SCALING AT ALL
    g2d.setFont(font);

    // Center text horizontally
    FontMetrics metrics = g2d.getFontMetrics();
    int textWidth = metrics.stringWidth(text);

```

```

int textX = (width - textWidth) / 2;

// Calculate text Y position based on whether image is present
int textY;
if (wordImage != null) {
    textY = height * 3 / 4;
} else {
    textY = (height - metrics.getHeight()) / 2 + metrics.getAscent();
}

// NO AUTO-SCALING - draw text as is, even if it goes outside bounds
g2d.drawString(text, textX, textY);
g2d.dispose();

return image;
}

/***
 * Sets the inverse transform matrix of the object
 * @param inverseTransform Matrix4 inverse transform
 */
@Override
public void setObjectTransform(Matrix4 inverseTransform) {
    if (inverseTransform != null) {
        this.objectInverseTransform = inverseTransform;
    } else {
        this.objectInverseTransform = new Matrix4();
    }
}

public void setTriangleEtc(boolean nbool) {
    this.isTriangleEtc = nbool;
}

/***
 * Returns the color at the given world point on the surface
 * Uses planar UV mapping on XY plane similar to
TransparentPNGMaterial
 * @param point World space point on surface

```

```

* @param normal Surface normal
* @param light Light source
* @param viewerPos Viewer position
* @return Color with alpha channel
*/
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    if(texture == null) {
        setTransparency(1.0);
        return new Color(0, 0, 0, 0);
    }

    Point3 local = objectInverseTransform.transformPoint(point);

    double u = 0.0;
    double v = 0.0;

    if(isTriangleEtc) {
        u = local.x + 0.5;
        v = local.z + 0.5;
    } else {
        u = (local.x + 1.0) * 0.5;
        v = (1.0 - (local.y + 1.0) * 0.5);
    }

    // Apply UV scale and offset
    double scaledU = u / uScale + uOffset;
    double scaledV = v / vScale + vOffset;

    double finalU, finalV;

    if(isRepeatTexture) {
        finalU = scaledU - Math.floor(scaledU);
        finalV = scaledV - Math.floor(scaledV);
    } else {
        if(scaledU < 0.0 || scaledU > 1.0 || scaledV < 0.0 || scaledV > 1.0) {
            setTransparency(1.0);
            return new Color(0, 0, 0, 0);
        }
    }
}

```

```
        }
        finalU = scaledU;
        finalV = scaledV;
    }

    int px = (int) (finalU * (texture.getWidth() - 1));
    int py = (int) (finalV * (texture.getHeight() - 1));

    int argb = texture.getRGB(px, py);

    int alpha = (argb >> 24) & 0xFF;
    int red = (argb >> 16) & 0xFF;
    int green = (argb >> 8) & 0xFF;
    int blue = argb & 0xFF;

    if (alpha > 5) {
        setTransparency(0.0);
        return new Color(red, green, blue, 255);
    }

    setTransparency(1.0);
    return new Color(0, 0, 0, 0);
}

/***
 * Returns reflectivity of the material
 * @return 0.0 (non-reflective)
 */
@Override
public double getReflectivity() {
    return 0.0;
}

/***
 * Returns index of refraction
 * @return 1.0 (no refraction)
 */
@Override
public double getIndexOfRefraction() {
```

```
    return 1.0;
}

/***
 * Returns transparency of the material
 * @return transparency value (0.0 for opaque, 1.0 for fully transparent)
 */
@Override
public double getTransparency() {
    return transparency;
}

private void setTransparency(double tnw) {
    this.transparency = tnw;
}

/***
 * Gets the horizontal texture offset
 * @return U offset value
 */
public double getUOffset() {
    return uOffset;
}

/***
 * Sets the horizontal texture offset
 * @param uOffset U offset value
 */
public void setUOffset(double uOffset) {
    this.uOffset = uOffset;
}

/***
 * Gets the vertical texture offset
 * @return V offset value
 */
public double getVOffset() {
    return vOffset;
}
```

```
/**  
 * Sets the vertical texture offset  
 * @param vOffset V offset value  
 */  
public void setVOffset(double vOffset) {  
    this.vOffset = vOffset;  
}  
  
/**  
 * Gets the horizontal texture scale factor  
 * @return U scale factor  
 */  
public double getUScale() {  
    return uScale;  
}  
  
/**  
 * Sets the horizontal texture scale factor  
 * @param uScale U scale factor  
 */  
public void setUScale(double uScale) {  
    this.uScale = (uScale > 0.0) ? uScale : 1.0;  
}  
  
/**  
 * Gets the vertical texture scale factor  
 * @return V scale factor  
 */  
public double getVScale() {  
    return vScale;  
}  
  
/**  
 * Sets the vertical texture scale factor  
 * @param vScale V scale factor  
 */  
public void setVScale(double vScale) {  
    this.vScale = (vScale > 0.0) ? vScale : 1.0;
```

```
}

/***
 * Checks if texture repeating is enabled
 * @return true if texture repeating is enabled, false otherwise
 */
public boolean isRepeatTexture() {
    return isRepeatTexture;
}

/***
 * Sets whether texture repeating is enabled
 * @param repeat true to enable repeating, false to disable
 */
public void setRepeatTexture(boolean repeat) {
    this.isRepeatTexture = repeat;
}

/***
 * Gets the rendered text
 * @return The text displayed on this material
 */
public String getText() {
    return text;
}

/***
 * Gets the foreground color
 * @return Text color
 */
public Color getForegroundColor() {
    return foregroundColor;
}

/***
 * Gets the background color
 * @return Background color
 */
public Color getBackgroundColor() {
```

```
    return backgroundColor;
}

/***
 * Gets the font used for rendering
 * @return The font
 */
public Font getFont() {
    return font;
}

/***
 * Checks if gradient is enabled
 * @return true if gradient is enabled
 */
public boolean isGradientEnabled() {
    return gradientEnabled;
}

/***
 * Gets the gradient end color
 * @return Gradient color
 */
public Color getGradientColor() {
    return gradientColor;
}

/***
 * Gets the optional word image
 * @return The word image or null if not set
 */
public BufferedImage getWordImage() {
    return wordImage;
}

/***
 * Gets the texture width
 * @return Texture width in pixels
 */
```

```
public int getWidth() {
    return width;
}

/***
 * Sets the texture width and regenerates the texture
 * @param width New texture width
 */
public void setWidth(int width) {
    this.width = width;
    regenerateTexture();
}

/***
 * Gets the texture height
 * @return Texture height in pixels
 */
public int getHeight() {
    return height;
}

/***
 * Sets the texture height and regenerates the texture
 * @param height New texture height
 */
public void setHeight(int height) {
    this.height = height;
    regenerateTexture();
}

/***
 * Sets both texture dimensions and regenerates the texture
 * @param width New texture width
 * @param height New texture height
 */
public void setSize(int width, int height) {
    this.width = width;
    this.height = height;
    regenerateTexture();
```

```
}

/***
 * Regenerates the texture with new text
 * @param newText New text to render
 */
public void setText(String newText) {
    this.text = newText;
    regenerateTexture();
}

/***
 * Regenerates the texture with new colors
 * @param newForeground New text color
 * @param newBackground New background color
 */
public void setColors(Color newForeground, Color newBackground) {
    this.foregroundColor = newForeground;
    this.backgroundColor = newBackground;
    regenerateTexture();
}

/***
 * Regenerates the texture with new font
 * @param newFont New font to use
 */
public void setFont(Font newFont) {
    this.font = newFont;
    regenerateTexture();
}

/***
 * Regenerates the texture with gradient settings
 * @param useGradient Whether to use gradient
 * @param newGradientColor Gradient end color
 */
public void setGradient(boolean useGradient, Color newGradientColor) {
    this.gradientEnabled = useGradient;
    this.gradientColor = newGradientColor;
}
```

```

        regenerateTexture();
    }

    /**
     * Regenerates the texture with new word image
     * @param newWordImage New word image to display (null for text
only)
 */
public void setWordImage(BufferedImage newWordImage) {
    this.wordImage = newWordImage;
    regenerateTexture();
}

/**
 * Regenerates the texture with all current settings
 * Useful when multiple properties change and you want to update once
 */
public void regenerateTexture() {
    this.texture = createTextImage(text, foregroundColor, backgroundColor,
font,
        gradientEnabled, gradientColor, wordImage, width, height);
}

}

/**
// Only text
Material ekmekMaterial = new WordMaterial("Ekmek");

// Text + image
BufferedImage breadImage = ImageIO.read(new File("bread.png"));
Material ekmekMaterial = new WordMaterial("Ekmek", Color.BLACK,
new Color(0x00000000, true), new Font("Arial", Font.BOLD, 36),
breadImage);

// Gradient + image
Material brodMaterial = new WordMaterial("Brød", Color.BLUE,
new Color(0x00000000, true), new Font("Arial", Font.BOLD, 42),
Color.CYAN, breadImage);

```

```
*/
```

```
/***
```

```
// Seffaflik icin daima ARGB formatinda yazin:  
new Color(0x00000000, true); // Tam seffaf  
new Color(0x80ffffffff, true); // %50 seffaf beyaz
```

```
// RGB formatinda yazacaksaniz hasAlpha=false kullanin:  
new Color(0xffffffff); // Opak beyaz  
new Color(0x000000); // Opak siyah
```

```
*/
```

```
/***
```

```
Triangle t1 {  
point1 = P(-0.5, 0.0, -0.5);  
point2 = P(0.5, 0.0, -0.5);  
point3 = P(0.0, 2.0, 0.0);  
material = lambert;  
}
```

```
Triangle t2 {  
point1 = P(0.5, 0.0, -0.5);  
point2 = P(0.5, 0.0, 0.5);  
point3 = P(0.0, 2.0, 0.0);  
material = lambert;  
}
```

```
Triangle t3 {  
point1 = P(0.5, 0.0, 0.5);  
point2 = P(-0.5, 0.0, 0.5);  
point3 = P(0.0, 2.0, 0.0);  
material = wordMaterial  
}
```

```
Triangle t4 {  
point1 = P(-0.5, 0.0, 0.5);  
point2 = P(-0.5, 0.0, -0.5);  
point3 = P(0.0, 2.0, 0.0);  
material = lambert;
```

```

}

*/
// =====
// File: /net/elenamurat/material/RadialGradientMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

public class RadialGradientMaterial implements Material {
    private final Color centerColor;
    private final Color edgeColor;

    public RadialGradientMaterial(Color centerColor, Color edgeColor) {
        this.centerColor = centerColor;
        this.edgeColor = edgeColor;
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPos) {
        double dist = Math.sqrt(point.x * point.x + point.y * point.y);
        dist = Math.min(dist, 1.0);

        int r = (int)(centerColor.getRed() * (1 - dist) + edgeColor.getRed() *
dist);
        int g = (int)(centerColor.getGreen() * (1 - dist) +
edgeColor.getGreen() * dist);
        int b = (int)(centerColor.getBlue() * (1 - dist) + edgeColor.getBlue() *
dist);

        r = ColorUtil.clampColorValue (r);

```

```
        g = ColorUtil.clampColorValue (g);
        b = ColorUtil.clampColorValue (b);

        return new Color(r, g, b);
    }

@Override
public double getReflectivity() {
    return 0.0;
}

@Override
public double getIndexOfRefraction() {
    return 1.0;
}

@Override
public double getTransparency() {
    return 0.0; //opaque
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

// =====
// File: /net/elenamurat/material/LavaFlowMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;
```

```
public class LavaFlowMaterial implements Material {  
    private final Color hotColor;  
    private final Color coolColor;  
    private final double flowSpeed;  
    private Matrix4 objectInverseTransform;  
    private double time;  
    private final double reflectivity=0.1;  
  
    public LavaFlowMaterial(Color hotColor, Color coolColor,  
        double flowSpeed, Matrix4 invTransform) {  
        // Initialize colors by clamping values  
        this.hotColor = new Color(  
            clamp(hotColor.getRed(), 0, 255),  
            clamp(hotColor.getGreen(), 0, 255),  
            clamp(hotColor.getBlue(), 0, 255)  
        );  
        this.coolColor = new Color(  
            clamp(coolColor.getRed(), 0, 255),  
            clamp(coolColor.getGreen(), 0, 255),  
            clamp(coolColor.getBlue(), 0, 255)  
        );  
        this.flowSpeed = flowSpeed;  
        this.objectInverseTransform = invTransform;  
    }  
  
    public void update(double deltaTime) {  
        time += deltaTime * flowSpeed;  
    }  
}
```

```
@Override  
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4 ();  
    this.objectInverseTransform = tm;  
}
```

```
@Override  
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light  
light, Point3 viewPos) {
```

```

Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
Vector3 normal =
objectInverseTransform.transformNormal(worldNormal).normalize();

// Noise functions (guaranteed in 0-1 range)
double nx = localPoint.x * 2 + time;
double ny = localPoint.y * 3;
double nz = time * 0.5;
double noise1 = fract(Math.sin(nx) * 43758.5453);
double noise2 = fract(Math.cos(ny) * 12578.1459);
double pattern = clamp(Math.sin(noise1 * Math.PI * 3) *
Math.cos(noise2 * Math.PI * 2), -1, 1) * 0.5 + 0.5;

// Temperature variation (guaranteed in 0-1 range)
double temp = clamp(0.5 + 0.5 * Math.sin(localPoint.y * 5 - time * 0.3),
0, 1);

// Base color calculation (guaranteed in 0-255 range)
Color baseColor = ColorUtil.blendColors(
    coolColor,
    hotColor,
    temp * (0.5 + 0.5 * pattern)
);

// Lighting calculations (guaranteed in 0-255 range)
Vector3 lightDir = light.getPosition().subtract(worldPoint).normalize();
double NdotL = clamp(normal.dot(lightDir), 0, 1);
double intensity = clamp(light.getIntensityAt(worldPoint), 0, 1);

Color directLight = ColorUtil.multiplyColors(
    baseColor,
    light.getColor(),
    NdotL * intensity
);

// Emissive glow (guaranteed in 0-255 range)
double glow = clamp(Math.pow(temp * 0.8 + 0.2, 3) * (0.7 + 0.3 *
pattern), 0, 1);

```

```

Color emissive = ColorUtil.multiplyColors(
    hotColor,
    light.getColor(),
    glow * 0.5
);

// Combine with clamping (guaranteed in 0-255 range)
return ColorUtil.add(directLight, emissive);
}

// Helper methods
private static int clamp(int value, int min, int max) {
    return Math.max(min, Math.min(max, value));
}

private static double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

private static double fract(double value) {
    return value - Math.floor(value);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.4; }
@Override public double getTransparency() { return 0.0; }
}

/**
 * Camera setup (common for all materials)
rayTracer.setCameraPosition(new Point3(0, 0, 5));
rayTracer.setLookAt(new Point3(0, 0, 0));
rayTracer.setUpVector(new Vector3(0, 1, 0));
rayTracer.setFov(45.0);

// Ambient light (soft lighting for entire scene)
scene.addLight(new ElenaMuratAmbientLight(
    new Color(200, 220, 255), // Bluish ambient
    0.3                  // Intensity
)

```

```

));
// 1. Create sphere
Sphere lavaSphere = new Sphere(1.2);
lavaSphere.setTransform(Matrix4.translate(new Vector3(2, -0.5, -6)));
// 2. Create material
LavaFlowMaterial lavaMat = new LavaFlowMaterial(
new Color(255, 80, 0), // Lava hot color
new Color(100, 0, 0), // Cooled lava color
0.8, // Flow speed
lavaSphere.getInverseTransform());
// 3. Assign material
lavaSphere.setMaterial(lavaMat);
// 4. Add to scene
scene.addShape(lavaSphere);
// 5. Update for animation in render loop
void renderLoop() {
double deltaTime = 0.016; // ~60 FPS
lavaMat.update(deltaTime);
// ... rendering operations
}
// 6. Proper lighting
scene.addLight(new MuratPointLight(
new Point3(0, 3, 0),
new Color(255, 100, 50), // Orange light
3.0
));
*/
// =====
// File: /net/elenamurat/material/WaterRippleMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class WaterRippleMaterial implements Material {
    private final Color waterColor;
    private final double waveSpeed;
    private final double reflectivity;
    private Matrix4 objectInverseTransform;
    private double time;

    public WaterRippleMaterial(Color waterColor, double waveSpeed,
        Matrix4 invTransform) {
        this(waterColor, waveSpeed, 0.4, invTransform);
    }

    public WaterRippleMaterial(Color waterColor, double waveSpeed,
        double reflectivity, Matrix4 invTransform) {
        this.waterColor = waterColor;
        this.waveSpeed = waveSpeed;
        this.reflectivity = Math.max(0, Math.min(1, reflectivity));
        this.objectInverseTransform = invTransform;
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectInverseTransform = tm;
    }

    public void update(double deltaTime) {
        time += deltaTime * waveSpeed;
    }
}
```

```

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
    // 1. Wave deformation
    Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
    double wave = calculateWaveEffect(localPoint);

    // 2. Normal perturbation
    Vector3 normal = perturbNormal(worldNormal, localPoint, wave);

    // 3. Light calculations
    Vector3 lightDir = light.getPosition().subtract(worldPoint).normalize();
    Vector3 viewDir = viewPos.subtract(worldPoint).normalize();
    Color lightColor = light.getColor();
    double intensity = light.getIntensityAt(worldPoint);

    // 4. Fresnel effect
    double fresnel = calculateFresnel(normal, viewDir);

    // 5. Specular reflection
    double specular = calculateSpecular(normal, lightDir, viewDir);

    // 6. Color blending
    Color base = ColorUtil.multiplyColor(waterColor, 0.7 + wave * 0.1);
    Color reflection = ColorUtil.multiplyColor(lightColor, fresnel *
reflectivity * intensity);
    Color highlight = ColorUtil.multiplyColor(lightColor, specular *
intensity);

    return ColorUtil.combineColors(base, reflection, highlight);
}

private double calculateWaveEffect(Point3 p) {
    return 0.3 * (
        Math.sin(p.x * 3 + time) +
        Math.cos(p.z * 4 + time * 1.3) +
        Math.sin(p.x * 5 + p.z * 7 + time * 0.7)
    ) / 3.0;
}

```

```

}

private Vector3 perturbNormal(Vector3 original, Point3 p, double wave)
{
    double dx = 0.5 * Math.cos(p.x * 5 + time);
    double dz = 0.5 * Math.sin(p.z * 5 + time);
    return new Vector3(
        original.x + dx,
        original.y,
        original.z + dz
    ).normalize();
}

private double calculateFresnel(Vector3 normal, Vector3 viewDir) {
    return Math.pow(1.0 - Math.max(0, normal.dot(viewDir)), 5);
}

private double calculateSpecular(Vector3 normal, Vector3 lightDir,
Vector3 viewDir) {
    Vector3 halfway = lightDir.add(viewDir).normalize();
    return Math.pow(Math.max(0, normal.dot(halfway)), 128);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.33; } // Water's IOR
@Override public double getTransparency() { return 0.3; }
}

/**
 * Scene setup
 */
Scene scene = new Scene();

// 1. Ceramic sphere
Sphere ceramicSphere = new Sphere(1.0);
ceramicSphere.setTransform(Matrix4.translate(new Vector3(-2, 0, -5)));
ceramicSphere.setMaterial(new DamaskCeramicMaterial(
    new Color(240, 240, 240), // White
    new Color(70, 70, 70) // Gray
);

```

```

50.0,           // Shine
ceramicSphere.getInverseTransform()
));
scene.addShape(ceramicSphere);

// 2. Water sphere
WaterRippleMaterial waterMat = new WaterRippleMaterial(
new Color(80, 180, 220), // Water blue
0.5,           // Wave speed
Matrix4.translate(new Vector3(2, 0, -5)).inverse()
);
Sphere waterSphere = new Sphere(1.0);
waterSphere.setTransform(Matrix4.translate(new Vector3(2, 0, -5)));
waterSphere.setMaterial(waterMat);
scene.addShape(waterSphere);

// 3. Light source
scene.addLight(new MuratPointLight(
new Point3(0, 5, 0),
new Color(255, 220, 180), // Warm white
2.0
));

// In render loop
double deltaTime = 0.016; // ~60 FPS
waterMat.update(deltaTime);
*/

```

```

package net.elena.murat.material;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.util.Random;

```

```
import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

/**
 * GhostTextMaterial - Transparent ghost material with text and image
 * texture support
 * Maintains original constructor signatures while adding dielectric
 * properties
 */
public class GhostTextMaterial implements Material {

    // Original texture properties (DO NOT CHANGE NAMES)
    private final String word;
    private final Color textColor;
    private final Color gradientColor;
    private final String gradientType;
    private final String fontFamily;
    private final int fontStyle;
    private final int fontSize;
    private final int uOffset;
    private final int vOffset;
    private final BufferedImage imageObject;
    private final int imageWidth;
    private final int imageHeight;
    private final int imageUOffset;
    private final int imageVOffset;
    private final BufferedImage texture;

    // New transparent material properties
    //private final double baseTransparency;
    private double transparency;
    private final double reflectivity;
    private final double indexOfRefraction;
    private final Color surfaceColor;

    private Matrix4 objectTransform;
    private final Random random;
```

```

/**
 * ORIGINAL CONSTRUCTOR - 15 parameters
 */
public GhostTextMaterial(String word, Color textColor, Color
gradientColor,
String gradientType,
String fontFamily, int fontStyle, int fontSize,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset) {

// Texture properties
this.word = convertToNorwegianText(word).replaceAll("_", " ");
this.textColor = textColor;
this.gradientColor = gradientColor;
this.gradientType = gradientType != null ? gradientType : "horizontal";
this.fontFamily = fontFamily.replaceAll("_", " ");
this.fontStyle = fontStyle;
this.fontSize = fontSize;
this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

// Default transparent properties
this.transparency = 0.8;
this.reflectivity = 0.1;
this.indexOfRefraction = 1.5;
this.surfaceColor = new Color(0, 0, 0, (int)(this.transparency * 255));

//this.baseTransparency = this.transparency;

this.random = new Random();
this.objectTransform = new Matrix4().identity();
this.texture = createTexture();
}

```

```

/***
 * NEW CONSTRUCTOR - 18 parameters (with transparency
properties)
 */
public GhostTextMaterial(String word, Color textColor, Color
gradientColor,
String gradientType,
String fontFamily, int fontStyle, int fontSize,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset,
double transparency, double reflectivity, double ior) {

// Texture properties
this.word = convertToNorwegianText(word).replaceAll("_", " ");
this.textColor = textColor;
this.gradientColor = gradientColor;
this.gradientType = gradientType != null ? gradientType : "horizontal";
this.fontFamily = fontFamily.replaceAll("_", " ");
this.fontStyle = fontStyle;
this.fontSize = fontSize;
this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

// Transparent properties
this.transparency = Math.max(0, Math.min(1, transparency));
this.reflectivity = Math.max(0, Math.min(1, reflectivity));
this.indexOfRefraction = Math.max(1.0, ior);
this.surfaceColor = new Color(0, 0, 0, (int)(this.transparency * 255));

//this.baseTransparency = this.transparency;

this.random = new Random();

```

```

this.objectTransform = new Matrix4().identity();
this.texture = createTexture();
}

/**
 * SIMPLIFIED CONSTRUCTOR - 5 parameters
 */
public GhostTextMaterial(String word, Color textColor,
String fontFamily, int fontStyle, int fontSize) {
this(word, textColor, null, "horizontal",
fontFamily, fontStyle, fontSize, 0, 0,
null, 0, 0, 0, 0);
}

/**
 * Creates the texture (ORIGINAL METHOD - DO NOT CHANGE)
 */
private BufferedImage createTexture() {
final int size = 1024;
BufferedImage texture = new BufferedImage(size, size,
BufferedImage.TYPE_INT_ARGB);
Graphics2D g2d = texture.createGraphics();

g2d.setComposite(AlphaComposite.Clear);
g2d.clearRect(0, 0, size, size);
g2d.setComposite(AlphaComposite.SrcOver);

// Full transparent bg garantized for eviting black circle
// alrededor sphere
//g2d.setBackground(new Color(0, 0, 0, 0));
//g2d.clearRect(0, 0, size, size);

g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

if (imageObject != null) {
int imgX = ((size - imageWidth) / 2) + imageUOffset - (imageWidth *

```

```

2);
    int imgY = ((size - imageHeight) / 2) + imageVOffset;
    g2d.drawImage(imageObject, imgX, imgY, imageWidth, imageHeight,
null);
}
}

Font font;
try {
    font = new Font(fontFamily, fontStyle, fontSize);
} catch (Exception e) {
    font = new Font("Arial", fontStyle, fontSize);
}
g2d.setFont(font);

FontMetrics fm = g2d.getFontMetrics();
int textWidth = fm.stringWidth(word);
int textHeight = fm.getHeight();
int ascent = fm.getAscent();

int x = ((size - textWidth) / 2) + uOffset - (textWidth * 2);
int y = ((size - textHeight) / 2) + (ascent * 2) + (textHeight / 3) +
vOffset;

if (gradientColor != null) {
    GradientPaint gradient = createGradient(x, y - ascent, textWidth,
textHeight);
    g2d.setPaint(gradient);
} else {
    g2d.setColor(textColor);
}

g2d.drawString(word, x, y);
g2d.dispose();

return texture;
}

private GradientPaint createGradient(float x, float y, float width, float
height) {

```

```
switch (gradientType.toLowerCase()) {
    case "vertical":
        return new GradientPaint(x, y, textColor, x, y + height/2,
gradientColor, true);
    case "diagonal":
        return new GradientPaint(x, y, textColor, x + width/3, y + height/5,
gradientColor, true);
    case "horizontal":
    default:
        return new GradientPaint(x, y, textColor, x + width/3, y,
gradientColor, true);
}
```

```
public static String convertToNorwegianText(String input) {
    if (input == null || input.isEmpty()) {
        return input;
    }
```

```
    String result = input;
    result = result.replace("AE", "\u00C6");
    result = result.replace("O/", "\u00D8");
    result = result.replace("A0", "\u00C5");
    result = result.replace("ae", "\u00E6");
    result = result.replace("o/", "\u00F8");
    result = result.replace("a0", "\u00E5");
```

```
    return result;
}
```

```
/***
 * NEW: getColorAt method with proper alpha blending for transparent
material
 */
```

```
@Override
```

```
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
```

```
    Color backgroundColor = calculateBackgroundColor(point, normal,
light, viewerPoint);
```

```

Point3 localPoint = objectTransform.inverse().transformPoint(point);
Color textureColor = getTextureColor(localPoint, normal);

if (textureColor.getAlpha() == 0) {
    //this.transparency = this.transparency * 1.2; // %20
    return backgroundColor;
}

float textureAlpha = textureColor.getAlpha() / 255.0f;

float r = (textureColor.getRed() / 255.0f * textureAlpha) +
(backgroundColor.getRed() / 255.0f * (1 - textureAlpha));
float g = (textureColor.getGreen() / 255.0f * textureAlpha) +
(backgroundColor.getGreen() / 255.0f * (1 - textureAlpha));
float b = (textureColor.getBlue() / 255.0f * textureAlpha) +
(backgroundColor.getBlue() / 255.0f * (1 - textureAlpha));

// Alpha: background transparency + texture visibility
float a = Math.max(backgroundColor.getAlpha() / 255.0f,
textureAlpha);

//this.transparency = baseTransparency;

return new Color(r, g, b, a);
}

private Color calculateBackgroundColor(Point3 point, Vector3 normal,
Light light, Point3 viewerPoint) {
    int a = (int)(transparency * 255.0);
    a = ColorUtil.clampColorValue(a);
    return new Color(0, 0, 0, a);
}

private Color getTextureColor(Point3 localPoint, Vector3 worldNormal)
{
    if (texture == null) return new Color(textureColor.getRed(),
textureColor.getGreen(), textureColor.getBlue(), 255);
}

```

```

Vector3 dir = worldNormal.normalize();

double u = 0.5 + Math.atan2(dir.z, dir.x) / (2 * Math.PI);
double v = 0.5 - Math.asin(dir.y) / Math.PI;

u = (u % 1.0 + 1.0) % 1.0;
v = (v % 1.0 + 1.0) % 1.0;

// Kenarları atla - merkeze yakın pikselleri kullan
double edgeMargin = 0.05;
if (u < edgeMargin || u > 1.0 - edgeMargin || v < edgeMargin || v > 1.0 - edgeMargin) {
    return new Color(0, 0, 0, 0); // Kenarlar şeffaf
}

int texX = (int) (u * texture.getWidth());
int texY = (int) (v * texture.getHeight());

texX = Math.max(0, Math.min(texture.getWidth() - 1, texX));
texY = Math.max(0, Math.min(texture.getHeight() - 1, texY));

int rgb = texture.getRGB(texX, texY);
return new Color(rgb, true);
}

// Material interface methods
@Override
public void setObjectTransform(Matrix4 tm) {
    this.objectTransform = (tm != null) ? tm : new Matrix4();
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

// Getters for transparent properties
public Color getSurfaceColor() {
    return surfaceColor;
}

@Override
public String toString() {
    return String.format("GhostTextMaterial[text='%s', trans=%,.2f, refl=%,.2f, ior=%,.2f]",
        word, transparency, reflectivity, indexOfRefraction);
}

}
```

```
// =====
// File: /net/lena/murat/material/CalligraphyRuneMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CalligraphyRuneMaterial implements Material {
    private final Color parchmentColor;
    private final Color inkColor;
    private final Color goldLeafColor;
    private final double writingIntensity;
```

```

private Matrix4 objectTransform;

private final double ambientCoeff = 0.45;
private final double diffuseCoeff = 0.8;
private final double specularCoeff = 0.18;
private final double shininess = 25.0;
private final double reflectivity = 0.1;
private final double ior = 1.6;
private final double transparency = 0.0;

public CalligraphyRuneMaterial() {
    this(new Color(0xF5, 0xDE, 0xB3), new Color(0x2F, 0x4F, 0x4F), new
Color(0xFF, 0xD7, 0x00), 0.65);
}

public CalligraphyRuneMaterial(Color parchmentColor, Color inkColor,
Color goldLeafColor, double writingIntensity) {
    this.parchmentColor = parchmentColor;
    this.inkColor = inkColor;
    this.goldLeafColor = goldLeafColor;
    this.writingIntensity = Math.max(0, Math.min(1, writingIntensity));
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateWritingPattern(objectPoint,
worldNormal);

```

```

LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
if (props == null) return surfaceColor;

Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(goldLeafColor, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateWritingPattern(Point3 point, Vector3 normal) {
    double x = point.x * 15.0;
    double y = point.y * 15.0;
    double z = point.z * 15.0;

    // Islamic calligraphy flowing patterns
    double calligraphy1 = Math.sin(x * 1.2 + Math.cos(y * 0.8) * 2.0);
    double calligraphy2 = Math.abs(Math.cos(x * 1.5 + y * 1.0) +
Math.sin(y * 1.2 + z * 0.7));
    double calligraphy3 = Math.sin(x * 2.0 + y * 1.5) * Math.cos(y * 1.0 + z
* 0.5);

    // Viking rune angular patterns
    double rune1 = Math.abs(Math.sin(x * 2.5) * Math.cos(y * 2.0));
}

```

```

double rune2 = (Math.floor(x * 0.8) + Math.floor(y * 0.8)) % 2.5;
double rune3 = Math.abs(Math.sin(x * 3.0 + y * 1.7) + Math.cos(y *
2.3));

// Cultural fusion writing pattern
double calligraphyWeight = 0.5 * writingIntensity;
double runeWeight = 0.5 * writingIntensity;

double combinedPattern = (calligraphy1 * 0.2 + calligraphy2 * 0.15 +
calligraphy3 * 0.15) * calligraphyWeight +
(rune1 * 0.2 + rune2 * 0.15 + rune3 * 0.15) * runeWeight;

double normalizedPattern = (combinedPattern + 1.0) * 0.5;

// View-dependent effect for gold leaf
Vector3 viewDir = new Vector3(0, 0, 1); // Simple view direction
double viewEffect = Math.abs(viewDir.dot(normal)) * 0.5 + 0.5;

if (normalizedPattern < 0.4) {
    // Parchment background with aging
    double ageEffect = normalizedPattern / 0.4;
    return ColorUtil.darkenColor(parchmentColor, ageEffect * 0.2);
} else if (normalizedPattern < 0.7) {
    // Ink writing (both calligraphy and runes)
    double intensity = (normalizedPattern - 0.4) / 0.3;
    Color writingInk = ColorUtil.darkenColor(inkColor, intensity * 0.3);
    return ColorUtil.addColorVariation(writingInk, intensity);
} else {
    // Gold leaf accents and decorations
    double intensity = (normalizedPattern - 0.7) / 0.3;
    Color gold = ColorUtil.blendColors(goldLeafColor,
    ColorUtil.lightenColor(goldLeafColor, 0.3), intensity);
    return ColorUtil.multiplyColors(gold, Color.WHITE, viewEffect);
}
}

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```
}
```

```
@Override
public double getIndexOfRefraction() {
    return ior;
}
```

```
@Override
public double getTransparency() {
    return transparency;
}
```

```
}
```

```
// =====
// File: /net/elenamurat/material/ObsidianMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

public class ObsidianMaterial implements Material {

    private double edgeSharpness;
    private double reflectivity;
    private Matrix4 objectTransform;

    public ObsidianMaterial() {
        this.edgeSharpness = 0.3;
        this.reflectivity = 0.04;
    }
}
```

```
public ObsidianMaterial(double edgeSharpness, double reflectivity) {
    this.edgeSharpness = Math.max(0, Math.min(1, edgeSharpness));
    this.reflectivity = Math.max(0, Math.min(0.1, reflectivity));
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Color baseColor = Color.BLACK;

    Vector3 viewDir = new Vector3(point, viewerPoint).normalize();
    double dotProduct = normal.dot(viewDir);

    if (edgeSharpness > 0) {
        double edgeFactor = Math.pow(1.0 - Math.abs(dotProduct), 2.0) *
edgeSharpness;

        int edgeValue = (int) (30 * edgeFactor);
        edgeValue = ColorUtil.clampColorValue(edgeValue);

        return new Color(edgeValue, edgeValue, edgeValue);
    }

    return baseColor;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    this.objectTransform = tm;
}

@Override
public double getTransparency() {
    return 0.02;
}

@Override
public double getIndexOfRefraction() {
```

```
    return 1.48;
}

@Override
public double getReflectivity() {
    return this.reflectivity;
}

public double getEdgeSharpness() {
    return edgeSharpness;
}

public void setEdgeSharpness(double edgeSharpness) {
    this.edgeSharpness = Math.max(0, Math.min(1, edgeSharpness));
}

public double getReflectivityValue() {
    return reflectivity;
}

public void setReflectivity(double reflectivity) {
    this.reflectivity = Math.max(0, Math.min(0.1, reflectivity));
}

}
```

```
// =====
// File: /net/elenamurat/material/PhongElenaMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

public class PhongElenaMaterial implements Material {
```

```

private final Color diffuseColor;
private final double reflectivity;
private final double shininess;
private final double ambientCoefficient;

public PhongElenaMaterial(Color diffuseColor, double reflectivity,
double shininess) {
    this(diffuseColor, reflectivity, shininess, 0.1);
}

public PhongElenaMaterial(Color diffuseColor, double reflectivity,
double shininess, double ambientCoefficient) {
    this.diffuseColor = diffuseColor;
    this.reflectivity = Math.max(0, Math.min(1, reflectivity));
    this.shininess = Math.max(1, shininess);
    this.ambientCoefficient = Math.max(0, Math.min(1,
ambientCoefficient));
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // 1. Get light properties
    Color lightColor = light.getColor();
    double intensity = light.getIntensityAt(worldPoint);
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

    // 2. Diffuse component (including light color)
    double NdotL = Math.max(0, worldNormal.dot(lightDir));
    int r = (int)(diffuseColor.getRed() * NdotL *
(lightColor.getRed()/255.0) * intensity);
    int g = (int)(diffuseColor.getGreen() * NdotL *
(lightColor.getGreen()/255.0) * intensity);
    int b = (int)(diffuseColor.getBlue() * NdotL *
(lightColor.getBlue()/255.0) * intensity);

    // 3. Specular component (including light color)
    if(NdotL > 0) {
        Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();

```

```

Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specular = Math.pow(RdotV, shininess) * reflectivity *
intensity;

r += (int)(lightColor.getRed() * specular);
g += (int)(lightColor.getGreen() * specular);
b += (int)(lightColor.getBlue() * specular);
}

// 4. Ambient component (NOT including light color)
r += (int)(diffuseColor.getRed() * ambientCoefficient);
g += (int)(diffuseColor.getGreen() * ambientCoefficient);
b += (int)(diffuseColor.getBlue() * ambientCoefficient);

// 5. Color clamping
return new Color(
    Math.min(255, r),
    Math.min(255, g),
    Math.min(255, b)
);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

/**
// 1. Basic usage
Material bluePhong = new PhongElenaMaterial(
    Color.BLUE, // Main color
    0.3,        // Reflectivity coefficient (0-1)
    50          // Shininess

```

```

);

// 2. Version with ambient adjustment
Material redPhong = new PhongElenaMaterial(
Color.RED,
0.5,
100,
0.2 // Ambient coefficient
);

// 3. For metallic effect with high reflectivity
Material goldPhong = new PhongElenaMaterial(
new Color(255, 215, 0), // Gold color
0.8, // Strong reflection
150 // High shininess
);

// Red light + Blue material = Purple tones
Light redLight = new MuratPointLight(new Point3(0,2,0), Color.RED,
1.5);
Material bluePhong = new PhongElenaMaterial(Color.BLUE, 0.3, 50);

// Green light + Red material = Yellowish tones
Light greenLight = new ElenaDirectionalLight(new Vector3(0,-1,0),
Color.GREEN, 1.2);
Material redPhong = new PhongElenaMaterial(Color.RED, 0.5, 100);
*/

```

```

// =====
// File: /net/elenamurat/material/GradientChessMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

```

```
import net.elena.murat.util.ColorUtil;

/**
 * Material that adds color gradients to a standard checkerboard pattern
 */
public class GradientChessMaterial implements Material {
    private final Color baseColor1;
    private final Color baseColor2;
    private final double squareSize;
    private Matrix4 objectInverseTransform;

    // Light properties
    private final double ambientCoefficient;
    private final double diffuseCoefficient;
    private final double specularCoefficient;
    private final double shininess;
    private final double reflectivity;
    private final double ior;
    private final double transparency;
    private final Color specularColor;

    public GradientChessMaterial(Color baseColor1, Color baseColor2,
        double squareSize,
        double ambient, double diffuse, double specular,
        double shininess, double reflectivity,
        double ior, double transparency,
        Matrix4 objectInverseTransform) {
        this.baseColor1 = baseColor1;
        this.baseColor2 = baseColor2;
        this.squareSize = Math.max(0.1, squareSize);
        this.ambientCoefficient = ambient;
        this.diffuseCoefficient = diffuse;
        this.specularCoefficient = specular;
        this.shininess = shininess;
        this.reflectivity = reflectivity;
        this.ior = ior;
        this.transparency = transparency;
        this.objectInverseTransform = objectInverseTransform;
        this.specularColor = Color.WHITE;
    }
}
```

```

}

// Simple constructor
public GradientChessMaterial(Color baseColor1, Color baseColor2,
    double squareSize,
    Matrix4 objectInverseTransform) {
    this(baseColor1, baseColor2, squareSize,
        0.1, 0.7, 0.2, 15.0, 0.1, 1.3, 0.05,
        objectInverseTransform);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

/**
 * Checkerboard pattern with applied gradient
 */
private Color getPatternColor(double u, double v) {
    // Normalize UV coordinates
    double normalizedU = (u / squareSize) % 2.0;
    double normalizedV = (v / squareSize) % 2.0;

    // Basic checkerboard pattern
    boolean isColor1 = ((int)normalizedU + (int)normalizedV) % 2 == 0;

    // Gradient factors (0-1 range)
    double uGradient = normalizedU % 1.0;
    double vGradient = normalizedV % 1.0;

    // Base color selection
    Color baseColor = isColor1 ? baseColor1 : baseColor2;
    Color targetColor = isColor1 ? baseColor2 : baseColor1;

    // Apply diagonal gradient
    double gradientFactor = (uGradient + vGradient) / 2.0;
}

```

```

    return ColorUtil.blendColors(baseColor, targetColor, gradientFactor);
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        return Color.BLACK;
    }

    // Transform to object space
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal()
    .transformVector(worldNormal).normalize();

    // UV mapping by dominant normal axis
    double absX = Math.abs(localNormal.x);
    double absY = Math.abs(localNormal.y);
    double absZ = Math.abs(localNormal.z);

    double u, v;
    if (absX > absY && absX > absZ) {
        u = localPoint.y;
        v = localPoint.z;
    } else if (absY > absX && absY > absZ) {
        u = localPoint.x;
        v = localPoint.z;
    } else {
        u = localPoint.x;
        v = localPoint.y;
    }

    // Get pattern color
    Color baseColor = getPatternColor(u, v);

    // Lighting calculations (Phong model)
    Color lightColor = light.getColor();
}

```

```

double attenuatedIntensity = 0.0;

// Ambient component
int rAmbient = (int)(baseColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
int gAmbient = (int)(baseColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
int bAmbient = (int)(baseColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);

if (light instanceof ElenaMuratAmbientLight) {
    return new Color(
        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

// Light direction handling
Vector3 lightDirection;
if (light instanceof MuratPointLight) {
    MuratPointLight pLight = (MuratPointLight) light;
    lightDirection = pLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = pLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof ElenaDirectionalLight) {
    ElenaDirectionalLight dLight = (ElenaDirectionalLight) light;
    lightDirection = dLight.getDirection().negate().normalize();
    attenuatedIntensity = dLight.getIntensity();
} else if (light instanceof PulsatingPointLight) {
    PulsatingPointLight ppLight = (PulsatingPointLight) light;
    lightDirection =
        ppLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = ppLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof SpotLight) {
    SpotLight sLight = (SpotLight) light;
    lightDirection = sLight.getDirectionAt(worldPoint);
    attenuatedIntensity = sLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BioluminescentLight) {
    BioluminescentLight bLight = (BioluminescentLight) light;
}

```

```

lightDirection = bLight.getDirectionAt(worldPoint);
attenuatedIntensity = bLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BlackHoleLight) {
BlackHoleLight bhLight = (BlackHoleLight) light;
lightDirection = bhLight.getDirectionAt(worldPoint);
attenuatedIntensity = bhLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof FractalLight) {
FractalLight fLight = (FractalLight) light;
lightDirection = fLight.getDirectionAt(worldPoint);
attenuatedIntensity = fLight.getAttenuatedIntensity(worldPoint);
}
else {
    return Color.BLACK;
}

// Diffuse component
double NdotL = Math.max(0, worldNormal.dot(lightDirection));
int rDiffuse = (int)(baseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int)(baseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int)(baseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

// Specular component
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectionVec = lightDirection.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectionVec.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

int rSpecular = (int)(specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
int gSpecular = (int)(specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
int bSpecular = (int)(specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

// Combine components
int finalR = Math.min(255, rAmbient + rDiffuse + rSpecular);

```

```

int finalG = Math.min(255, gAmbient + gDiffuse + gSpecular);
int finalB = Math.min(255, bAmbient + bDiffuse + bSpecular);

return new Color(finalR, finalG, finalB);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }

}

/***
Material chessMat = new GradientChessMaterial(
new Color(50, 50, 200), // Blue
new Color(200, 50, 50), // Red
1.0, // Square size
plane.getInverseTransform()
);

Material pastelMat = new GradientChessMaterial(
new Color(240, 180, 220), // Pink
new Color(180, 220, 240), // Blue
0.8, // Smaller squares
plane.getInverseTransform()
);
*/

```

```

// =====
// File: /net/elenamurat/material/PlatinumMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

```

```
import net.elena.murat.util.ColorUtil;

public class PlatinumMaterial implements Material {
    private static final Color BASE_COLOR = new Color(160, 158, 155);
    private static final Color COLD_SPECULAR = new Color(180, 200,
255);
    private static final Color WARM_SPECULAR = new Color(255, 230,
190);
    private static final double IOR = 2.05;
    private static final double REFLECTIVITY = 0.6;
    private static final double SHININESS = 80.0;
    private static final double METALLIC_DIFFUSE = 0.25;

    private Matrix4 objectInverseTransform;
    private final double specularBalance;

    public PlatinumMaterial(Matrix4 objectInverseTransform) {
        this(objectInverseTransform, 0.35);
    }

    public PlatinumMaterial(Matrix4 objectInverseTransform, double
specularBalance) {
        this.objectInverseTransform = new Matrix4(objectInverseTransform);
        this.specularBalance = Math.max(0, Math.min(1, specularBalance));
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectInverseTransform = tm;
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
        // Transform normal to object space
        Vector3 normal =
objectInverseTransform.inverseTransposeForNormal()
        .transformVector(worldNormal).normalize();
```

```

// Get light properties
LightProperties lightProps = LightProperties.getLightProperties(light,
worldPoint);
if (lightProps == null) return BASE_COLOR;

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
double NdotL = Math.max(0.1, normal.dot(lightProps.direction));

// Fresnel effect
double fresnel = 0.04 + 0.96 * Math.pow(1 - Math.max(0,
viewDir.dot(normal)), 5);

// Specular calculations
Color specularColor = ColorUtil.blendColors(COLD_SPECULAR,
WARM_SPECULAR, specularBalance);
Vector3 halfVec = lightProps.direction.add(viewDir).normalize();
double NdotH = Math.max(0, normal.dot(halfVec));
double specular = Math.pow(NdotH, SHININESS) * REFLECTIVITY
* fresnel;

// Apply light color and intensity
Color lightColor = lightProps.color;
double intensity = lightProps.intensity;

// Base diffuse component
int r = (int)(BASE_COLOR.getRed() * METALLIC_DIFFUSE *
NdotL *
(lightColor.getRed()/255.0) * intensity);
int g = (int)(BASE_COLOR.getGreen() * METALLIC_DIFFUSE *
NdotL *
(lightColor.getGreen()/255.0) * intensity);
int b = (int)(BASE_COLOR.getBlue() * METALLIC_DIFFUSE *
NdotL *
(lightColor.getBlue()/255.0) * intensity);

// Add specular component
r += (int)(specularColor.getRed() * specular *
(lightColor.getRed()/255.0) * intensity);

```

```
    g += (int)(specularColor.getGreen() * specular *
    (lightColor.getGreen()/255.0) * intensity);
    b += (int)(specularColor.getBlue() * specular *
    (lightColor.getBlue()/255.0) * intensity);

    return new Color(
        clamp(r, 0, 255),
        clamp(g, 0, 255),
        clamp(b, 0, 255)
    );
}
```

```
private int clamp(int value, int min, int max) {
    return Math.max(min, Math.min(max, value));
}
```

```
@Override
public double getReflectivity() {
    return REFLECTIVITY;
}
```

```
@Override
public double getIndexOfRefraction() {
    return IOR;
}
```

```
@Override
public double getTransparency() {
    return 0.0;
}

}
```

```
// =====
// File: /net/elenamurat/material/pbr/CopperPBRMaterial.java
// =====
```

```
package net.elenamurat.material.pbr;
```

```
import java.awt.Color;
import java.util.Random;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CopperPBRMaterial implements PBRCapableMaterial {
    private final Color albedo;
    private final double roughness;
    private final double oxidation;
    private final Random random = new Random();

    // Copper color constants
    public static final Color PURE_COPPER = new Color(184, 115, 51);
    public static final Color OXIDIZED_COPPER = new Color(100, 130, 90);

    // Constructors
    public CopperPBRMaterial() {
        this(0.0, 0.0); // Fully shiny, non-oxidized
    }

    public CopperPBRMaterial(double roughness, double oxidation) {
        this(PURE_COPPER, roughness, oxidation);
    }

    public CopperPBRMaterial(Color baseColor, double roughness, double oxidation) {
        this.albedo = baseColor;
        this.roughness = Math.max(0.0, Math.min(1.0, roughness));
        this.oxidation = Math.max(0.0, Math.min(1.0, oxidation));
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
                           Point3 viewerPos) {
        // 1. Oxidized color blend
```

```

Color baseColor = ColorUtil.lerp(
    albedo,
    OXIDIZED_COPPER,
    (float)oxidation
);

// 2. Vector calculations
Vector3 viewDir = new Vector3(point, viewerPos).normalize();
Vector3 lightDir = light.getDirectionTo(point).normalize();

// 3. Reflection vector (with roughness perturbation)
Vector3 reflected = Vector3.reflect(viewDir.negate(), normal);
if (roughness > 0) {
    Vector3 randomPerturbation =
        Vector3.randomInUnitSphere(random).scale(roughness);
    reflected = reflected.add(randomPerturbation).normalize();
}

// 4. Fresnel effect (special for metals)
double cosTheta = Math.max(0.001, normal.dot(viewDir));
double fresnel = 0.9 + 0.1 * Math.pow(1.0 - cosTheta, 5.0);

// 5. Specular calculation (GGX)
Vector3 halfway = viewDir.add(lightDir).normalize();
double specular = calculateGGX(normal, halfway, roughness) * fresnel;

// 6. Diffuse (reduced by oxidation)
double diffuse = Math.max(0.05, normal.dot(lightDir)) * (1.0 -
oxidation * 0.7);

// 7. Color composition
Color specularColor = ColorUtil.multiply(baseColor, (float)(specular *
2.0));
Color diffuseColor = ColorUtil.multiply(baseColor, (float)diffuse);

return ColorUtil.add(
    ColorUtil.scale(diffuseColor, 0.3),
    ColorUtil.scale(specularColor, 1.0 - oxidation * 0.5)
);

```

```

}

private double calculateGGX(Vector3 normal, Vector3 halfway, double roughness) {
    double alpha = roughness * roughness;
    double NdotH = Math.max(0, normal.dot(halfway));
    double denom = NdotH * NdotH * (alpha - 1.0) + 1.0;
    return alpha / (Math.PI * denom * denom);
}

// PBR Properties
@Override public Color getAlbedo() {
    return ColorUtil.lerp(albedo, OXIDIZED_COPPER, (float)oxidation);
}
@Override public double getRoughness() {
    return roughness + oxidation * 0.3; // Oxidation increases roughness
}
@Override public double getMetalness() {
    return 1.0 - oxidation * 0.5; // Oxidation reduces metallic property
}
@Override public MaterialType getMaterialType() {
    return oxidation > 0.7 ? MaterialType.DIELECTRIC :
MaterialType.METAL;
}
@Override public double getReflectivity() {
    return 0.9 - oxidation * 0.6;
}
@Override public double getIndexOfRefraction() {
    return 1.0 + oxidation * 0.5;
}
@Override public double getTransparency() {
    return 0.0;
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

}

```

```

/***
// 1. Shiny pure copper
Material pureCopper = new CopperPBRMaterial(0.05, 0.0);

// 2. Slightly oxidized (old copper pipe)
Material agedCopper = new CopperPBRMaterial(0.3, 0.4);

// 3. Green oxidized copper (statue)
Material oxidizedCopper = new CopperPBRMaterial(
new Color(80, 130, 80), // Greenish oxidation
0.6, // High roughness
0.9 // Full oxidation
);

// 4. Custom colored copper alloy
Material customCopper = new CopperPBRMaterial(
new Color(200, 120, 60), // Gold-copper blend
0.2,
0.1
);
*/

```

```

// =====
// File: /net/elenamurat/material/pbr/MarblePBRMaterial.java
// =====

```

```

package net.elena.murat.material.pbr;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;
import net.elena.murat.util.NoiseUtil;

public class MarblePBRMaterial implements PBRCapableMaterial {
    private final Color baseColor;

```

```

private final Color veinColor;
private final double veinScale;
private final double veinContrast;
private final double roughness;
private final double reflectivity;
private final double veinIntensity;

public MarblePBRMaterial() {
    this(new Color(230, 226, 220), new Color(100, 100, 100),
        15.0, 0.7, 0.3, 0.1, 1.0);
}

public MarblePBRMaterial(Color baseColor, Color veinColor, double
veinScale,
    double veinContrast, double roughness, double reflectivity,
    double veinIntensity) {
    this.baseColor = baseColor;
    this.veinColor = veinColor;
    this.veinScale = Math.max(1.0, veinScale);
    this.veinContrast = Math.max(0.1, Math.min(1.0, veinContrast));
    this.roughness = Math.max(0.01, Math.min(1.0, roughness));
    this.reflectivity = Math.max(0.0, Math.min(1.0, reflectivity));
    this.veinIntensity = Math.max(0.5, Math.min(2.0, veinIntensity));
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Advanced vein pattern (3D Perlin noise)
    Point3 scaledPoint = new Point3(
        point.x * veinScale,
        point.y * veinScale,
        point.z * veinScale * 0.5
    );

    double noise = NoiseUtil.turbulence(scaledPoint, 4);
    double veins = Math.pow(Math.sin(noise * Math.PI * 3) * 0.5 + 0.5,
        veinContrast * 10);
    veins *= veinIntensity;
}

```

```

// 2. Color blending (with gamma correction)
Color marbleColor = ColorUtil.lerp(
    ColorUtil.gammaCorrect(baseColor, 0.9f),
    ColorUtil.gammaCorrect(veinColor, 0.8f),
    (float)Math.min(0.9, veins) // Maximum vein intensity
);

// 3. Lighting calculations
Vector3 lightDir = light.getDirectionTo(point).normalize();
double NdotL = Math.max(0.4, normal.dot(lightDir)); // Minimum 0.4
brightness

// 4. Diffuse
Color diffuseColor = ColorUtil.multiply(marbleColor, (float)(NdotL *
1.3));

// 5. Specular (GGX approximation)
Vector3 viewDir = new Vector3(point, viewerPos).normalize();
Vector3 halfway = viewDir.add(lightDir).normalize();
double NdotH = Math.max(0.0, normal.dot(halfway));
double alpha = roughness * roughness;
double denominator = NdotH * NdotH * (alpha * alpha - 1.0) + 1.0;
double specular = (alpha * alpha) / (Math.PI * denominator *
denominator);

// 6. Final result
return ColorUtil.add(
    diffuseColor,
    ColorUtil.scale(Color.WHITE, specular * reflectivity * 1.5)
);
}

// PBR Properties
@Override public Color getAlbedo() { return baseColor; }
@Override public double getRoughness() { return roughness; }
@Override public double getMetalness() { return 0.0; }
@Override public MaterialType getMaterialType() { return
MaterialType.DIELECTRIC; }

```

```
@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.5; }
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

public MarblePBRMaterial withVeinIntensity(double intensity) {
    return new MarblePBRMaterial(baseColor, veinColor, veinScale,
        veinContrast, roughness, reflectivity, intensity);
}

/***
Material material = new MarblePBRMaterial(
    new Color(230, 226, 220), // Base color
    new Color(100, 100, 100), // Vein color
    15.0, // Vein scale (larger = more frequent)
    0.7, // Vein contrast (0.1-1.0)
    0.3, // Roughness (0.01-1.0)
    0.1, // Reflectivity (0.0-1.0)
    1.0 // Vein intensity (0.5-2.0)
);
*/

```

```
// =====
// File: /net/elenamurat/material/pbr/WoodPBRMaterial.java
// =====
```

```
package net.elena.murat.material.pbr;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;
```

```

public class WoodPBRMaterial implements PBRCapableMaterial {
    private final Color woodColor1, woodColor2;
    private final double tileSize;
    private final double roughness;
    private final double specularScale;
    private boolean isAlternateTile = false;

    public WoodPBRMaterial() {
        this(new Color(160, 110, 60), new Color(130, 90, 50), 0.5, 0.3, 1.5);
    }

    public WoodPBRMaterial(Color color1, Color color2, double tileSize,
                           double roughness, double specularScale) {
        this.woodColor1 = color1;
        this.woodColor2 = color2;
        this.tileSize = tileSize;
        this.roughness = roughness;
        this.specularScale = specularScale;
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
                           Point3 viewerPos) {
        // 1. Checkerboard pattern
        int tileX = (int)(point.x / tileSize) % 2;
        int tileZ = (int)(point.z / tileSize) % 2;
        isAlternateTile = (tileX + tileZ) % 2 == 0;

        // 2. Base color selection
        Color baseColor = isAlternateTile ? woodColor1 : woodColor2;

        // 3. Light calculations (Checkerboard's ambient/diffuse logic)
        Vector3 lightDir = light.getDirectionTo(point).normalize();
        double diffuse = Math.max(0.5, normal.dot(lightDir)); // Min 0.5
        brightness guarantee

        // 4. Wood texture (grain effect)
        double grain = 1.0 + Math.sin(point.x * 30) * 0.2;
    }
}

```

```

Color woodColor = ColorUtil.multiply(baseColor, (float)grain);

// 5. Specular (PBR)
Vector3 viewDir = new Vector3(point, viewerPos).normalize();
Vector3 halfway = viewDir.add(lightDir).normalize();
double specular = Math.pow(Math.max(0, normal.dot(halfway)), 50) *
specularScale;

// 6. RESULT: Checkerboard's brightness guarantee + Wood texture
return ColorUtil.add(
    ColorUtil.multiply(woodColor, (float)diffuse),
    ColorUtil.multiply(Color.WHITE, (float)(specular * 0.8)))
);

}

// PBR Properties
@Override public Color getAlbedo() { return isAlternateTile ?
woodColor1 : woodColor2; }
@Override public double getRoughness() { return roughness; }
@Override public double getMetalness() { return 0.0; }
@Override public MaterialType getMaterialType() { return
MaterialType.DIELECTRIC; }
@Override public double getReflectivity() { return 0.3; }
@Override public double getIndexOfRefraction() { return 1.53; }
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

/**
// 1. STRONG DIRECT light
scene.addLight(new MuratPointLight(
new Point3(2, 10, 3), // FROM ABOVE (reduces shadows)
new Color(255, 250, 240), // Near white
5.0 // Full power
));

```

```
// 2. OMNIDIRECTIONAL light (Ambient++)
scene.addLight(new ElenaMuratAmbientLight(
new Color(255, 240, 230),
1.5 // 1.5x intensity (more than normal)
));
```

```
// 3. GROUND BOUNCE LIGHT
scene.addLight(new MuratPointLight(
new Point3(0, -0.5, 0), // From ground upwards
new Color(200, 190, 180),
2.0
));
```

```
EMShape ground = new Plane();
ground.setTransform(
new Matrix4()
.rotateX(Math.toRadians(-90)) // HORIZONTAL
.translate(0, -1, 0) // Positioning
);
ground.setMaterial(new WoodPBRMaterial());
```

```
// CAMERA (Show full ground)
camera.setPosition(new Point3(3, 1.5, 4));
```

```
// Try these values in material constructor:
new WoodPBRMaterial(
new Color(180, 140, 90), // Light color
new Color(150, 100, 60), // Dark color
0.8, // Large tiles
0.2, // Very low roughness (shiny)
2.0 // High specular
);
*/
```

```
// =====
// File: /net/elenamurat/material/pbr/HolographicPBRMaterial.java
// =====
```

```
package net.elena.murat.material.pbr;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;
import net.elena.murat.util.MathUtil;

public class HolographicPBRMaterial implements PBRCapableMaterial {

    private final Color baseColor;
    private final double rainbowSpeed;
    private final double scanLineDensity;
    private final double glitchIntensity;
    private final double timeOffset;
    private final double distortionFactor;
    private final double dataDensity;

    // Special effect modes
    public enum HologramMode {
        STANDARD, CYBERPUNK, MATRIX, QUANTUM
    }

    public HolographicPBRMaterial() {
        this(new Color(80, 255, 255, 150), 2.5, 25.0, 0.3, 0.0, 0.5, 10.0);
    }

    public HolographicPBRMaterial(Color baseColor, double rainbowSpeed,
        double scanLineDensity,
        double glitchIntensity, double timeOffset,
        double distortionFactor, double dataDensity) {
        this.baseColor = baseColor;
        this.rainbowSpeed = rainbowSpeed;
        this.scanLineDensity = scanLineDensity;
        this.glitchIntensity = glitchIntensity;
        this.timeOffset = timeOffset;
        this.distortionFactor = distortionFactor;
        this.dataDensity = dataDensity;
    }
}
```

```
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    double time = System.currentTimeMillis() * 0.001 + timeOffset;

    // 1. Space-Time Distortion
    Point3 distortedPoint = applySpaceTimeDistortion(point, time);

    // 2. Dynamic Color Spectrum
    Color spectralColor = calculateSpectralColor(distortedPoint, time);

    // 3. Holographic Scan Lines
    double scanLineEffect = calculateScanLines(distortedPoint, time);

    // 4. Quantum Noise
    double quantumNoise = calculateQuantumNoise(distortedPoint, time);

    // 5. Data Packet Effect (Matrix-style)
    Color dataEffect = calculateDataEffect(distortedPoint, time);

    // 6. Anisotropic Specular
    double anisotropicSpecular = calculateAnisotropicSpecular(point,
normal, light, viewerPos, time);

    // 7. Glitch Effect
    Color glitchEffect = applyGlitchEffects(spectralColor, point, time);

    // 8. Final Color Composition
    return composeFinalColor(
        glitchEffect,
        dataEffect,
        scanLineEffect,
        quantumNoise,
        anisotropicSpecular
    );
}
```

```

// --- Special Effect Methods ---
private Point3 applySpaceTimeDistortion(Point3 p, double time) {
    return new Point3(
        p.x + Math.sin(p.y * 0.5 + time) * distortionFactor * 0.1,
        p.y + Math.cos(p.x * 0.3 + time * 1.3) * distortionFactor * 0.1,
        p.z
    );
}

private Color calculateSpectralColor(Point3 p, double time) {
    double hue = (p.x * rainbowSpeed + time) % 1.0;
    double saturation = 0.7 + Math.sin(p.y * 3 + time) * 0.2;
    double brightness = 0.8 + Math.cos(p.z * 4 - time * 2) * 0.1;
    return Color.getHSBColor((float)hue, (float)saturation,
    (float)brightness);
}

private double calculateScanLines(Point3 p, double time) {
    double verticalLines = Math.sin(p.y * scanLineDensity + time) * 0.4 +
    0.6;
    double horizontalLines = Math.pow(Math.sin(p.x * scanLineDensity *
    0.3 + time * 0.7), 2);
    return verticalLines * horizontalLines;
}

private double calculateQuantumNoise(Point3 p, double time) {
    return MathUtil.random(p.x * 1000 + p.y * 100 + p.z * 10 + time) * 0.2;
}

private Color calculateDataEffect(Point3 p, double time) {
    if (MathUtil.random(p.z * 1000 + time) > 0.9) {
        double greenValue = 0.3 + MathUtil.random(p.x * 100) * 0.7;
        return new Color(0, (int)(greenValue * 255), 0, 100);
    }
    return new Color(0, 0, 0, 0);
}

private double calculateAnisotropicSpecular(Point3 p, Vector3 n, Light
light, Point3 v, double time) {

```

```

Vector3 h = light.getDirectionTo(p).add(new Vector3(v,
p)).normalize();
Vector3 tangent = new Vector3(
    Math.sin(p.y * 10 + time),
    0,
    Math.cos(p.y * 10 + time)
).normalize();
double dotTH = tangent.dot(h);
return Math.pow(1 - dotTH * dotTH, 10) * 2.0;
}

private Color applyGlitchEffects(Color base, Point3 p, double time) {
    if (MathUtil.random(p.x * 500 + p.y * 300 + time) < glitchIntensity *
0.1) {
        return ColorUtil.invert(base);
    }
    if (MathUtil.random(p.y * 400 + p.z * 200 + time * 2) < glitchIntensity *
0.05) {
        return new Color(
            base.getGreen(),
            base.getBlue(),
            base.getRed(),
            base.getAlpha()
        );
    }
    return base;
}

private Color composeFinalColor(Color base, Color data, double
scanLine, double noise, double specular) {
    // Base color + scan lines
    Color c1 = ColorUtil.multiply(base, (float)(scanLine + noise));

    // Add data effect
    Color c2 = ColorUtil.add(c1, data);

    // Add specular (white highlight)
    return ColorUtil.add(c2, ColorUtil.scale(Color.WHITE, specular));
}

```

```

// --- PBR Properties (Customized for Holographic Material) ---
@Override public Color getAlbedo() {
    return ColorUtil.setAlpha(baseColor, 100);
}
@Override public double getRoughness() { return 0.15; }
@Override public double getMetalness() { return 0.7; }
@Override public MaterialType getMaterialType() { return
MaterialType.ANISOTROPIC; }
@Override public double getReflectivity() { return 0.85; }
@Override public double getIndexOfRefraction() { return 1.15; }
@Override public double getTransparency() { return 0.4; }

@Override
public void setObjectTransform(Matrix4 tm) {

}

// --- Easy Configuration with Builder Pattern ---
public static class Builder {
    private Color baseColor = new Color(100, 200, 255, 150);
    private double rainbowSpeed = 2.0;
    private double scanLineDensity = 20.0;
    private double glitchIntensity = 0.3;
    private double timeOffset = 0.0;
    private double distortionFactor = 0.5;
    private double dataDensity = 10.0;

    public Builder withBaseColor(Color color) {
        this.baseColor = color;
        return this;
    }

    public Builder withCyberpunkStyle() {
        this.rainbowSpeed = 3.5;
        this.scanLineDensity = 30.0;
        this.glitchIntensity = 0.7;
        return this;
    }
}

```

```
public Builder withMatrixStyle() {
    this.baseColor = new Color(0, 255, 0, 100);
    this.dataDensity = 20.0;
    return this;
}

public HolographicPBRMaterial build() {
    return new HolographicPBRMaterial(
        baseColor, rainbowSpeed, scanLineDensity,
        glitchIntensity, timeOffset, distortionFactor, dataDensity
    );
}
}

// --- Usage Examples ---
public static void demo() {
    // Standard hologram
    HolographicPBRMaterial standard = new HolographicPBRMaterial();

    // Cyberpunk style
    HolographicPBRMaterial cyberpunk = new Builder()
        .withBaseColor(new Color(255, 50, 200, 180))
        .withCyberpunkStyle()
        .build();

    // Matrix style
    HolographicPBRMaterial matrix = new Builder()
        .withMatrixStyle()
        .build();
}

// =====
// File: /net/lena/murat/material/pbr/WaterPBRMaterial.java
// =====

package net.lena.murat.material.pbr;
```

```
import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.*;

public class WaterPBRMaterial implements PBRCapableMaterial {
    private final Color waterColor;
    private final double roughness;
    private final double waveIntensity;
    private final double murkiness;
    private final double foamThreshold;
    private double time;

    // Predefined water types
    public static final Color OCEAN_BLUE = new Color(10, 90, 130, 150);
    public static final Color TROPICAL_TEAL = new Color(0, 180, 200, 120);
    public static final Color MURKY_RIVER = new Color(70, 100, 80, 200);

    public WaterPBRMaterial() {
        this(OCEAN_BLUE, 0.05, 0.3, 0.1, 0.7);
    }

    public WaterPBRMaterial(Color waterColor, double roughness,
                           double waveIntensity, double murkiness,
                           double foamThreshold) {
        this.waterColor = waterColor;
        this.roughness = MathUtil.clamp(roughness, 0.001, 0.5);
        this.waveIntensity = MathUtil.clamp(waveIntensity, 0.0, 1.0);
        this.murkiness = MathUtil.clamp(murkiness, 0.0, 1.0);
        this.foamThreshold = MathUtil.clamp(foamThreshold, 0.5, 1.0);
        this.time = 0.0;
    }

    public void update(double deltaTime) {
        this.time += deltaTime * 0.5; // Animation speed
```

```

}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Wave deformation
    Vector3 waveNormal = calculateWaveNormal(point, normal);

    // 2. Light interactions
    Color lighting = calculateLighting(point, waveNormal, light,
viewerPos);

    // 3. Foam formation
    Color foam = calculateFoam(point, waveNormal);

    // 4. Depth effect
    Color depthEffect = calculateDepthEffect(point);

    // 5. Result
    return ColorUtil.add(lighting, ColorUtil.add(foam, depthEffect));
}

private Vector3 calculateWaveNormal(Point3 p, Vector3 originalNormal)
{
    // Gerstner waves
    double wave1 = Math.sin(p.x * 0.5 + p.z * 0.3 + time * 1.5) *
waveIntensity;
    double wave2 = Math.cos(p.x * 0.3 - p.z * 0.4 + time * 2.0) *
waveIntensity * 0.7;

    // Normal calculation
    double dx = 0.5 * Math.cos(p.x * 0.5 + p.z * 0.3 + time * 1.5) *
waveIntensity;
    double dz = 0.3 * Math.sin(p.x * 0.3 - p.z * 0.4 + time * 2.0) *
waveIntensity;

    return new Vector3(
        originalNormal.x - dx,
        originalNormal.y,

```

```

        originalNormal.z - dz
    ).normalize();
}

private Color calculateLighting(Point3 p, Vector3 normal, Light light,
Point3 viewPos) {
    // Fresnel effect
    double NdotV = Math.max(0.001, normal.dot(new Vector3(p,
viewPos).normalize()));
    double fresnel = MathUtil.fresnelSchlick(NdotV, 1.33); // Water IOR

    // Specular (GGX)
    Vector3 lightDir = light.getDirectionTo(p).normalize();
    Vector3 halfway = lightDir.add(new Vector3(p, viewPos)).normalize();
    double NdotH = normal.dot(halfway);
    double specular = MathUtil.ggxDistribution(NdotH, roughness) *
MathUtil.fresnelSchlick(NdotH, 1.33);

    // Base color
    Color base = ColorUtil.scale(waterColor, 1.0 - murkiness * 0.7);

    // Reflection
    Color reflection = ColorUtil.scale(Color.WHITE, fresnel * 2.0);

    return ColorUtil.add(base, reflection);
}

private Color calculateFoam(Point3 p, Vector3 normal) {
    // Foam at wave peaks
    double foam = Math.sin(p.x * 2.0 + time * 3.0) *
Math.cos(p.z * 1.5 + time * 2.5) *
waveIntensity;

    if (foam > foamThreshold) {
        double intensity = (foam - foamThreshold) / (1.0 - foamThreshold);
        return ColorUtil.scale(Color.WHITE, intensity * 0.8);
    }
    return new Color(0, 0, 0, 0);
}

```

```

private Color calculateDepthEffect(Point3 p) {
    // Increasing murkiness with depth
    double depthFactor = MathUtil.clamp(-p.y * 0.5, 0.0, 1.0);
    return ColorUtil.scale(waterColor, depthFactor * murkiness);
}

// --- PBR Properties ---
@Override public Color getAlbedo() {
    return ColorUtil.setAlpha(waterColor, 150);
}
@Override public double getRoughness() {
    return roughness + waveIntensity * 0.2; // Waves increase roughness
}
@Override public double getMetalness() { return 0.0; }
@Override public MaterialType getMaterialType() { return
MaterialType.TRANSPARENT; }
@Override public double getReflectivity() { return 0.9; }
@Override public double getIndexOfRefraction() { return 1.33; } // Water IOR
@Override public double getTransparency() { return 0.8 - murkiness *
0.3; }

@Override
public void setObjectTransform(Matrix4 tm) {

}

// --- Builder Pattern ---
public static class Builder {
    private Color waterColor = OCEAN_BLUE;
    private double roughness = 0.05;
    private double waveIntensity = 0.3;
    private double murkiness = 0.1;
    private double foamThreshold = 0.7;

    public Builder withTropicalColor() {
        this.waterColor = TROPICAL_TEAL;
        return this;
    }
}

```

```
public Builder withStormyWaves() {
    this.waveIntensity = 0.8;
    this.roughness = 0.2;
    return this;
}

public WaterPBRMaterial build() {
    return new WaterPBRMaterial(
        waterColor, roughness,
        waveIntensity, murkiness,
        foamThreshold
    );
}
}

// --- Usage Examples ---
public static void demo() {
    // Clear ocean
    WaterPBRMaterial ocean = new WaterPBRMaterial();

    // Tropical lagoon
    WaterPBRMaterial lagoon = new Builder()
        .withTropicalColor()
        .build();

    // Stormy sea
    WaterPBRMaterial stormyOcean = new Builder()
        .withStormyWaves()
        .build();
}

// =====
// File: /net/elenamurat/material/pbr/SilverPBRMaterial.java
// =====
```

```
package net.elena.murat.material.pbr;

import java.awt.Color;
import java.util.Random;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class SilverPBRMaterial implements PBRCapableMaterial {
    private final Color albedo;
    private final double roughness;
    private final double metalness;
    private final Random random = new Random();

    // Silver color variations
    public static final Color PURE_SILVER = new Color(192, 192, 192);
    public static final Color POLISHED_SILVER = new Color(220, 220, 230);
    public static final Color ANTIQUE_SILVER = new Color(170, 170, 180);

    public SilverPBRMaterial(double roughness) {
        this(POLISHED_SILVER, roughness, 1.0);
    }

    public SilverPBRMaterial(Color albedo, double roughness, double metalness) {
        this.albedo = ColorUtil.adjustSaturation(albedo, 0.9f);
        this.roughness = Math.max(0.01, Math.min(1.0, roughness));
        this.metalness = Math.max(0.7, Math.min(1.0, metalness)); // Silver
        // should have high metalness
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPos) {
        // 1. Vector calculations
        Vector3 viewDir = new Vector3(point, viewerPos).normalize();
```

```

Vector3 lightDir = light.getDirectionTo(point).normalize();

// 2. Optimized Fresnel for silver
double cosTheta = Math.max(0.001, normal.dot(viewDir));
double fresnel = 0.85 + 0.15 * Math.pow(1.0 - cosTheta, 5.0); // Silver
has high reflectivity

// 3. Reflection vector (less perturbation for silver)
Vector3 reflected = Vector3.reflect(viewDir.negate(), normal);
if (roughness > 0.001) {
    reflected = reflected.add(
        Vector3.randomInUnitSphere(random).scale(roughness * 0.5) // Less
scattering than gold
    ).normalize();
}

// 4. Special specular for silver
Vector3 halfway = viewDir.add(lightDir).normalize();
double specular = Math.max(0, normal.dot(halfway));
double specularIntensity = Math.pow(specular,
    20 + 80 * (1.0 - roughness) // Sharper reflections for silver
) * (1.0 + metalness * 1.8); // Less boost than gold

// 5. Metallic color component (preserving silver color)
Color metallicColor = ColorUtil.multiply(
    ColorUtil.lerp(albedo, Color.WHITE, 0.2f), // Slightly whitened
    (float)(specularIntensity * fresnel * 2.5) // Less boost than gold
);

// 6. Diffuse (very little for silver)
double diffuseFactor = Math.max(0.05, normal.dot(lightDir)) * (1.0 -
metalness * 0.9);
Color diffuseColor = ColorUtil.scale(
    albedo,
    diffuseFactor * 0.8 // Less diffuse than gold
);

// 7. Ambient (cool-toned)
Color ambientColor = ColorUtil.scale(

```

```

        ColorUtil.lerp(albedo, new Color(200, 210, 220), 0.3f), // Cool tone
        0.2 * (1.0 + metalness)
    );

    // 8. Final result
    Color finalColor = ColorUtil.add(
        ambientColor,
        ColorUtil.add(diffuseColor, metallicColor)
    );

    // Light intensity (less than gold)
    float intensity = (float)light.getIntensityAt(point) * 0.9f;
    return ColorUtil.adjustContrast(
        ColorUtil.multiply(finalColor, intensity),
        1.1f // Less contrast
    );
}

// --- PBR Properties ---
@Override public Color getAlbedo() { return albedo; }
@Override public double getRoughness() { return roughness; }
@Override public double getMetalness() { return metalness; }
@Override public MaterialType getMaterialType() { return
MaterialType.METAL; }
@Override public double getReflectivity() { return 0.85; } // Slightly less
than gold
@Override public double getIndexOfRefraction() { return 0.18; } //
Metallic IOR
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

/**
Material polishedSilver = new SilverPBRMaterial(0.1); // Polished silver
Material brushedSilver = new SilverPBRMaterial(0.3); // Brushed silver

```

```

// Ideal lighting for silver
scene.addLight(new MuratPointLight(
new Point3(2, 5, 2),
new Color(255, 250, 245), // Warm white
2.0
));
// For cool reflections
scene.addLight(new MuratPointLight(
new Point3(-1, 3, -1),
new Color(200, 220, 255), // Blue tint
0.8
));
// Lighting setup
scene.clearLights();
scene.addLight(new ElenaDirectionalLight(
new Vector3(-1, -0.5, -0.5).normalize(), // Side lighting
new Color(230, 235, 240),
1.5
));
*/
/***
javac -cp ..\bin\elenaRT.jar; SilverTest.java

java -cp ..\bin\elenaRT.jar; SilverTest 3
*/
// =====
// File: /net/elena/murat/material/pbr/PlasticPBRMaterial.java
// =====

package net.elena.murat.material.pbr;

import java.awt.Color;

```

```

import net.elena.murat.light.*;
import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/**
 * PlasticPBRMaterial represents a non-metallic, dielectric material like
plastic, wood, or painted surfaces.
 * It implements the PBRCapableMaterial interface for Physically Based
Rendering.
 *
 * This material uses a simplified PBR approach:
 * - Albedo defines the base color (diffuse reflection).
 * - Roughness controls the size and spread of specular highlights.
 * - It assumes a fixed Index of Refraction (IOR ~ 1.5) for common
plastics.
 * - No diffuse component for metals, but plastic always has one.
 *
 * The shading model combines Lambertian diffuse with a Phong-like
specular term,
 * modulated by roughness. This is a step towards microfacet theory
without full complexity.
 */

public class PlasticPBRMaterial implements PBRCapableMaterial {
    private final Color albedo;
    private final double roughness;
    private final double ior;
    private final double transparency;
    private final double reflectivity; // Base reflectivity, can be adjusted

    /**
     * Constructs a PlasticPBRMaterial with full parameters.
     *
     * @param albedo The base color of the material (diffuse response).
     * @param roughness The surface roughness (0.0 = smooth, 1.0 = rough).
     * @param reflectivity The base reflectivity coefficient (0.0-1.0).
     * @param ior The Index of Refraction (igual or greater than 1.0,
typically 1.4-1.6 for plastics).
     * @param transparency The transparency level (0.0 = opaque, 1.0 =
clear).

```

```

*/
public PlasticPBRMaterial(Color albedo, double roughness, double
reflectivity, double ior, double transparency) {
    this.albedo = albedo;
    this.roughness = clamp01(roughness);
    this.reflectivity = clamp01(reflectivity);
    this.ior = Math.max(1.0, ior);
    this.transparency = clamp01(transparency);
}

/***
 * Simplified constructor for common plastic materials.
 * Default: roughness=0.3, reflectivity=0.04 (Fresnel base), IOR=1.5,
opaque.
 *
 * @param albedo The base color of the plastic.
 */
public PlasticPBRMaterial(Color albedo) {
    this(albedo, 0.3, 0.04, 1.5, 0.0);
}

public PlasticPBRMaterial () {
    this(new Color (0.1F, 0.2F, 0.8F), 0.3, 0.04, 1.5, 0.0);
}

// --- PBRCapableMaterial Interface Implementation ---

@Override
public Color getAlbedo() {
    return albedo;
}

@Override
public double getRoughness() {
    return roughness;
}

@Override
public double getMetalness() {

```

```

    return 0.0; // Plastic is non-metallic
}

@Override
public MaterialType getMaterialType() {
    return MaterialType.PLASTIC;
}

// --- Material Interface Implementation ---

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // Ambient component is minimal
    if(light instanceof ElenaMuratAmbientLight) {
        int r = (int) (albedo.getRed() * 0.05);
        int g = (int) (albedo.getGreen() * 0.05);
        int b = (int) (albedo.getBlue() * 0.05);
        return new Color(Math.min(255, r), Math.min(255, g), Math.min(255,
b));
    }
}

// Get light direction and intensity
Vector3 lightDir = getLightDirection(light, point);
if (lightDir == null) return Color.BLACK;

double intensity = getLightIntensity(light, point);
if (intensity <= 0) return Color.BLACK;

// Normalize vectors
Vector3 N = normal.normalize();
Vector3 L = lightDir.normalize();
Vector3 V = viewerPos.subtract(point).normalize();
Vector3 H = L.add(V).normalize(); // Half-vector

// Diffuse component (Lambert)
double NdotL = Math.max(0.0, N.dot(L));
double diffuseFactor = NdotL;

```

```

// Specular component (Blinn-Phong style, roughness affects shininess)
double NdotH = Math.max(0.0, N.dot(H));
double shininess = 1.0 / Math.max(0.001, roughness * roughness); //
Higher roughness = lower shininess
double specFactor = Math.pow(NdotH, shininess * 128.0); // Scale for
visual plausibility

// Fresnel approximation (Schlick) for reflectivity based on angle
double F0 = (1.0 - ior) / (1.0 + ior);
F0 = F0 * F0; // Base reflectivity at normal incidence
double cosTheta = Math.max(0.0, V.dot(N));
double fresnel = F0 + (1.0 - F0) * Math.pow(1.0 - cosTheta, 5);

// Combine components
Color lightColor = light.getColor();
int rDiffuse = (int) (albedo.getRed() * diffuseFactor * intensity *
lightColor.getRed() / 255.0);
int gDiffuse = (int) (albedo.getGreen() * diffuseFactor * intensity *
lightColor.getGreen() / 255.0);
int bDiffuse = (int) (albedo.getBlue() * diffuseFactor * intensity *
lightColor.getBlue() / 255.0);

int rSpecular = (int) (lightColor.getRed() * specFactor * fresnel * 
intensity * 1.5);
int gSpecular = (int) (lightColor.getGreen() * specFactor * fresnel * 
intensity * 1.5);
int bSpecular = (int) (lightColor.getBlue() * specFactor * fresnel * 
intensity * 1.5);

int r = Math.min(255, rDiffuse + rSpecular);
int g = Math.min(255, gDiffuse + gSpecular);
int b = Math.min(255, bDiffuse + bSpecular);

return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```
}
```

```
@Override  
public double getIndexOfRefraction() {  
    return ior;  
}
```

```
@Override  
public double getTransparency() {  
    return transparency;  
}
```

```
@Override  
public void setObjectTransform(Matrix4 tm) {  
}
```

```
// --- Helper Methods ---
```

```
private double clamp01(double val) {  
    return Math.min(1.0, Math.max(0.0, val));  
}
```

```
private double getLightIntensity(Light light, Point3 point) {  
    if (light instanceof MuratPointLight) {  
        return ((MuratPointLight) light).getAttenuatedIntensity(point);  
    } else if (light instanceof ElenaDirectionalLight) {  
        return ((ElenaDirectionalLight) light).getIntensity();  
    } else if (light instanceof PulsatingPointLight) {  
        return ((PulsatingPointLight) light).getAttenuatedIntensity(point);  
    } else if (light instanceof SpotLight) {  
        return ((SpotLight) light).getAttenuatedIntensity(point);  
    } else if (light instanceof BioluminescentLight) {  
        return ((BioluminescentLight) light).getAttenuatedIntensity(point);  
    } else if (light instanceof BlackHoleLight) {  
        return ((BlackHoleLight) light).getAttenuatedIntensity(point);  
    } else if (light instanceof FractalLight) {  
        return ((FractalLight) light).getAttenuatedIntensity(point);  
    }  
    return 1.0;
```

```
}

private Vector3 getLightDirection(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight)
light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight)
light).getDirectionAt(point).normalize();
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getDirectionAt(point).normalize();
    }
    return null;
}

}

// =====
// File: /net/elenamurat/material/pbr/ChromePBRMaterial.java
// =====
```

```
package net.elena.murat.material.pbr;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.*;
```

```

public class ChromePBRMaterial implements PBRCapableMaterial {
    private final Color baseReflectance;
    private final double roughness;
    private final double anisotropy;
    private final double clearCoat;
    private final Color edgeTint;

    // Predefined chrome colors
    public static final Color MIRROR_CHROME = new Color(220, 230,
240);
    public static final Color BLACK_CHROME = new Color(70, 80, 90);
    public static final Color ROSE_CHROME = new Color(255, 200, 220);

    public ChromePBRMaterial() {
        this(MIRROR_CHROME, 0.02, 0.3, 1.0, new Color(150, 180, 255));
    }

    public ChromePBRMaterial(Color baseReflectance, double roughness,
    double anisotropy, double clearCoat, Color edgeTint) {
        this.baseReflectance = baseReflectance;
        this.roughness = MathUtil.clamp(roughness, 0.001, 0.5);
        this.anisotropy = MathUtil.clamp(anisotropy, 0.0, 1.0);
        this.clearCoat = MathUtil.clamp(clearCoat, 0.5, 1.5);
        this.edgeTint = edgeTint;
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
        // 1. Vector calculations
        Vector3 viewDir = new Vector3(point, viewerPos).normalize();
        Vector3 lightDir = light.getDirectionTo(point).normalize();
        Vector3 halfway = viewDir.add(lightDir).normalize();

        // 2. Fresnel effect (colored edges)
        double cosTheta = Math.max(0.001, normal.dot(viewDir));
        Color fresnelColor = calculateEdgeFresnel(cosTheta);
    }
}

```

```

// 3. Anisotropic reflection
double specular = calculateAnisotropicSpecular(normal, halfway,
lightDir, viewDir);

// 4. ClearCoat layer
double coatIntensity = calculateClearCoat(viewDir, normal);

// 5. Color composition
return composeFinalColor(fresnelColor, specular, coatIntensity);
}

private Color calculateEdgeFresnel(double cosTheta) {
    // Edge color with Schlick approximation
    double f0 = 0.8;
    double fresnel = f0 + (1 - f0) * Math.pow(1 - cosTheta, 5);

    return ColorUtil.lerp(
        baseReflectance,
        edgeTint,
        (float)(MathUtil.clamp((fresnel * 1.5), 0, 1)));
}

private double calculateAnisotropicSpecular(Vector3 normal, Vector3
halfway,
Vector3 lightDir, Vector3 viewDir) {
    // GGX anisotropic distribution
    Vector3 tangent = new Vector3(1, 0, 0); // Surface texture direction
    Vector3 bitangent = normal.cross(tangent);

    double aspect = Math.sqrt(1 - anisotropy * 0.9);
    double ax = Math.max(0.001, roughness * roughness / aspect);
    double ay = Math.max(0.001, roughness * roughness * aspect);

    double NdotH = normal.dot(halfway);
    double TdotH = tangent.dot(halfway);
    double BdotH = bitangent.dot(halfway);

    // Anisotropic GGX
    double denominator = (TdotH * TdotH) / (ax * ax) +

```

```

(BdotH * BdotH) / (ay * ay) +
NdotH * NdotH;
double distribution = 1.0 / (Math.PI * ax * ay * denominator *
denominator);

// Geometry function
double NdotL = normal.dot(lightDir);
double NdotV = normal.dot(viewDir);
double G = MathUtil.smithG1(NdotL, roughness) *
MathUtil.smithG1(NdotV, roughness);

return distribution * G / (4 * NdotL * NdotV);
}

private double calculateClearCoat(Vector3 viewDir, Vector3 normal) {
    // Secondary reflection layer
    double coatRoughness = 0.1;
    double NdotV = Math.max(0.001, normal.dot(viewDir));
    return MathUtil.fresnelSchlick(NdotV, 1.5) * clearCoat;
}

private Color composeFinalColor(Color base, double specular, double
coat) {
    // Base color (chrome reflects almost no diffuse)
    Color baseColor = ColorUtil.multiply(base, 0.1f);

    // Main reflection
    Color specColor = ColorUtil.scale(base, specular * 2.0);

    // ClearCoat layer (white reflection)
    Color coatColor = ColorUtil.scale(Color.WHITE, coat * 0.8);

    return ColorUtil.add(baseColor, ColorUtil.add(specColor, coatColor));
}

// --- PBR Properties ---
@Override public Color getAlbedo() {
    return ColorUtil.scale(baseReflectance, 0.05f); // Chrome reflects very
little albedo
}

```

```

}

@Override public double getRoughness() { return roughness; }
@Override public double getMetalness() { return 1.0; } // Fully metallic
@Override public MaterialType getMaterialType() { return
MaterialType.ANISOTROPIC; }
@Override public double getReflectivity() { return 0.95; }
@Override public double getIndexOfRefraction() { return 2.5; } // High
IOR for chrome
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

// --- Builder Pattern ---
public static class Builder {
    private Color baseReflectance = MIRROR_CHROME;
    private double roughness = 0.02;
    private double anisotropy = 0.3;
    private double clearCoat = 1.0;
    private Color edgeTint = new Color(150, 180, 255);

    public Builder withRoughness(double roughness) {
        this.roughness = roughness;
        return this;
    }

    public Builder withAnisotropy(double anisotropy) {
        this.anisotropy = anisotropy;
        return this;
    }

    public Builder asBlackChrome() {
        this.baseReflectance = BLACK_CHROME;
        this.edgeTint = new Color(100, 120, 150);
        return this;
    }

    public ChromePBRMaterial build() {
}

```

```

        return new ChromePBRMaterial(baseReflectance, roughness,
            anisotropy, clearCoat, edgeTint);
    }
}

// --- Usage Examples ---
public static void demo() {
    // Mirror-like chrome
    ChromePBRMaterial mirror = new ChromePBRMaterial();

    // Black chrome (custom settings)
    ChromePBRMaterial blackChrome = new Builder()
        .asBlackChrome()
        .withRoughness(0.05)
        .build();

    // Custom chrome (rose gold)
    ChromePBRMaterial roseGold = new ChromePBRMaterial(
        new Color(255, 200, 180), // Base color
        0.03,                  // Low roughness
        0.4,                   // Noticeable anisotropy
        1.2,                   // Thick clear coat
        new Color(255, 220, 180) // Warm edge color
    );
}

}

// =====
// File: /net/elenamurat/material/pbr/MaterialType.java
// =====

package net.elena.murat.material.pbr;

/**
 * Advanced PBR material types with hologram-specific extensions.
 * Includes spectral, temporal and quantum rendering modes.
 */

```

```
public enum MaterialType {  
    // Basic PBR Types  
    METAL,          // Traditional metallic surfaces  
    DIELECTRIC,     // Insulating materials (glass, plastic)  
    PLASTIC,        // Micro-surface distributed plastic  
    EMISSIVE,       // Self-illuminating surfaces  
    TRANSPARENT,    // Light-transmitting materials  
  
    // Holographic/Advanced Types  
    HOLOGRAM,       // Basic holographic surface  
    ANISOTROPIC,     // Directional reflection properties  
    SPECTRAL,       // Wavelength-based color distribution  
    VOLUMETRIC,      // Volumetric effects (fog, smoke)  
    QUANTUM,         // Quantum wave function effects  
    GLITCH,          // Digital distortion effects  
    BIOLUMINESCENT, // Biological illumination  
    PHANTOM          // Phantom image (partial intersection)  
};  
  
/**  
 * Checks if this material type requires temporal calculations.  
 */  
public boolean isTimeDependent() {  
    return this == HOLOGRAM ||  
        this == SPECTRAL ||  
        this == QUANTUM ||  
        this == GLITCH;  
}  
  
/**  
 * Checks if material has volumetric properties.  
 */  
public boolean isVolumetric() {  
    return this == VOLUMETRIC ||  
        this == PHANTOM;  
}  
  
/**  
 * Checks if material requires special light transport.  
 */
```

```

*/
public boolean needsSpecialLightHandling() {
    return this == ANISOTROPIC ||
    this == BIOLUMINESCENT;
}

/***
 * Suggested roughness range for material type.
 */
public double[] getSuggestedRoughnessRange() {
    switch(this) {
        case HOLOGRAM: return new double[]{0.05, 0.3};
        case ANISOTROPIC: return new double[]{0.1, 0.5};
        case GLITCH: return new double[]{0.2, 0.8};
        default: return new double[]{0.0, 1.0};
    }
}

}

// =====
// File: /net/elenamurat/material/pbr/GlassicTilePBRMaterial.java
// =====

package net.elena.murat.material.pbr;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class GlassicTilePBRMaterial implements PBRCapableMaterial {
    protected final Color tileColor;
    private final Color groutColor;
    private final double tileSize;
    private final double groutWidth;
    private final double tileRoughness;
}

```

```

private final double groutRoughness;

public static final Color DEFAULT_TILE_COLOR = new Color(245,
245, 255);
public static final Color DEFAULT_GROUT_COLOR = new Color(70,
70, 80);
public static final double DEFAULT_TILE_SIZE = 0.6;
public static final double DEFAULT_GROUT_WIDTH = 0.03;
public static final double DEFAULT_TILE_ROUGHNESS = 0.1;
public static final double DEFAULT_GROUT_ROUGHNESS = 0.6;

public GlassicTilePBRMaterial() {
    this(DEFAULT_TILE_COLOR, DEFAULT_GROUT_COLOR,
        DEFAULT_TILE_SIZE, DEFAULT_GROUT_WIDTH,
        DEFAULT_TILE_ROUGHNESS,
        DEFAULT_GROUT_ROUGHNESS);
}

public GlassicTilePBRMaterial(Color tileColor, Color groutColor,
    double tileSize, double groutWidth,
    double tileRoughness, double groutRoughness) {
    this.tileColor = ColorUtil.adjustSaturation(tileColor, 1.2f);
    this.groutColor = groutColor;
    this.tileSize = Math.max(0.1, tileSize);
    this.groutWidth = Math.max(0.01, Math.min(0.1, groutWidth));
    this.tileRoughness = clamp(tileRoughness, 0.01, 1.0);
    this.groutRoughness = clamp(groutRoughness, 0.01, 1.0);
}

//Metallic nice like glass material.
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Tile pattern
    double u = (point.x + 1000) % tileSize / tileSize;
    double v = (point.z + 1000) % tileSize / tileSize;
    boolean isGroutLocal = (u < groutWidth/tileSize || u > 1 -
groutWidth/tileSize ||
        v < groutWidth/tileSize || v > 1 - groutWidth/tileSize);
}

```

```

Color baseColor = isGroutLocal ? groutColor : tileColor;
double roughness = isGroutLocal ? groutRoughness : tileRoughness;

// Vectors
Vector3 lightDir = light.getDirectionTo(point).normalize();
Vector3 viewDir = viewerPos.subtract(point).normalize();
Vector3 halfway = lightDir.add(viewDir).normalize();

double NdotL = Math.max(0.0, normal.dot(lightDir));
double NdotV = Math.max(0.0, normal.dot(viewDir));
if (NdotL == 0.0) {
    return ColorUtil.multiply(baseColor, 0.1f); // Ambient
}

// Fresnel (Schlick)
double HdotV = Math.max(0.0, halfway.dot(viewDir));
double F0 = 0.04; // Dielectric
double fresnel = F0 + (1.0 - F0) * Math.pow(1.0 - HdotV, 5);

// --- GGX Distribution ---
double alpha = roughness * roughness;
double NdotH = Math.max(0.001, normal.dot(halfway));
double denom = NdotH * NdotH * (alpha * alpha - 1.0) + 1.0; // Fix:
alpha^2
double D = alpha * alpha / (Math.PI * denom * denom);

// --- Smith G (GGX) Shadowing ---
double k = roughness * roughness / 2.0;
double G1 = NdotL / (NdotL * (1 - k) + k);
double G2 = NdotV / (NdotV * (1 - k) + k);
double G = G1 * G2;

// Specular BRDF
double specular = (D * G * fresnel) / Math.max(0.001, 4.0 * NdotL *
NdotV);
specular = Math.min(specular, 10.0); // Prevent overflow

// Diffuse (Lambert)

```

```

double diffuse = NdotL;

float[] baseRGB = ColorUtil.getFloatComponents(baseColor);

// Albedo scale: (1 - F0)
Color diffuseColor = ColorUtil.createColor(
    baseRGB[0] * (1.0 - F0) * 255,
    baseRGB[1] * (1.0 - F0) * 255,
    baseRGB[2] * (1.0 - F0) * 255
);

// Diffuse
Color diffuseLight = ColorUtil.multiply(diffuseColor, (float) diffuse);

float F0f = (float) F0;
Color specularLight = ColorUtil.createColor(
    F0f * 255, F0f * 255, F0f * 255
);
specularLight = ColorUtil.multiply(specularLight, (float) specular);

// Ambient
Color ambient = ColorUtil.multiply(diffuseColor, 0.1f);

Color total = ColorUtil.add(diffuseLight, specularLight);
total = ColorUtil.add(total, ambient);

// Tonemapping: Reinhard (HDR → LDR)
float[] rgb = ColorUtil.getFloatComponents(total);
float exposure = 0.6f; // Bu değeri ayarla: 0.4–1.0
rgb[0] = rgb[0] * exposure;
rgb[1] = rgb[1] * exposure;
rgb[2] = rgb[2] * exposure;

// Reinhard tonemapping
rgb[0] = rgb[0] / (rgb[0] + 1.0f);
rgb[1] = rgb[1] / (rgb[1] + 1.0f);
rgb[2] = rgb[2] / (rgb[2] + 1.0f);

return ColorUtil.createColor(rgb[0] * 255, rgb[1] * 255, rgb[2] * 255);

```

```
}

private double calculateSpecular(Vector3 normal, Vector3 halfway,
                                double roughness, double fresnel) {
    // Bu metot artık kullanılmıyor, yukarıda inline yazdık
    return 0;
}

@Override
public Color getAlbedo() {
    return ColorUtil.blendColors(tileColor, groutColor, 0.7f);
}

@Override
public double getRoughness() {
    return (tileRoughness + groutRoughness) / 2.0;
}

@Override public double getMetalness() { return 0.0; }
@Override public MaterialType getMaterialType() { return
MaterialType.DIELECTRIC; }
@Override public double getReflectivity() { return 0.4; }
@Override public double getIndexOfRefraction() { return 1.52; }
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
    // Optional
}

private double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

}

// =====
// File: /net/elenamurat/material/pbr/GoldPBRMaterial.java
```

```
// =====
```

```
package net.elena.murat.material.pbr;

import java.awt.Color;
import java.util.Random;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.material.Material;
import net.elena.murat.util.ColorUtil;

public class GoldPBRMaterial implements PBRCapableMaterial {
    private final Color albedo;
    private final double roughness;
    private final double metalness;
    private final Random random = new Random();

    public GoldPBRMaterial(double roughness) {
        this(new Color(255, 215, 0), // Standard gold color (RGB)
             roughness,
             1.0); // Full metal
    }

    public GoldPBRMaterial(Color albedo, double roughness, double
metalness) {
        this.albedo = albedo;
        this.roughness = Math.max(0.0, Math.min(1.0, roughness));
        this.metalness = Math.max(0.0, Math.min(1.0, metalness));
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
        // 1. Vector calculations (optimized)
```

```

Vector3 viewDir = new Vector3(point, viewerPos).normalize();
Vector3 lightDir = light.getDirectionTo(point).normalize();

// 2. Enhanced Fresnel (for more vibrant edge reflections)
double cosTheta = Math.max(0.001, normal.dot(viewDir)); // Prevent
division by zero
double fresnel = Math.pow(1.0 - cosTheta, 5.0);
fresnel = (0.98 * metalness) + (0.02 * fresnel); // Stronger base
reflectivity in metals

// 3. Reflection vector (optimized perturbation)
Vector3 reflected = Vector3.reflect(viewDir.negate(), normal);
if (roughness > 0.001) { // Skip small roughness values
    reflected = reflected.add(
        Vector3.randomInUnitSphere(random).scale(roughness *
roughness) // roughness^2 looks more natural
    ).normalize();
}

// 4. Enhanced Specular (brighter highlights)
Vector3 halfway = viewDir.add(lightDir).normalize();
double specular = Math.max(0, normal.dot(halfway));
double specularIntensity = Math.pow(specular,
    16 + 64 * (1.0 - roughness) // Dynamic shininess
) * (1.0 + metalness * 2.0); // Specular boost in metals

// 5. Color components (for more vibrant colors)
Color metallicColor = ColorUtil.multiply(
    albedo,
    (float)(specularIntensity * fresnel * 3.0) // 3x boost
);

// 6. Optimized Diffuse (preserve color saturation)
double diffuseFactor = Math.max(0.1, normal.dot(lightDir)) * (1.0 -
metalness * 0.8);
Color diffuseColor = ColorUtil.scale(
    ColorUtil.gammaCorrect(albedo, 0.8F), // Saturation booster
    diffuseFactor * 1.2
);

```

```

// 7. Enhanced Ambient (preserve color temperature)
Color ambientColor = ColorUtil.scale(
    ColorUtil.lerp(albedo, Color.WHITE, 0.2f), // Slightly whitened
    0.15 * (1.0 + metalness) // Brighter ambient in metals
);

// 8. Result (with tone mapping)
Color finalColor = ColorUtil.add(
    ambientColor,
    ColorUtil.add(diffuseColor, metallicColor)
);

// Light intensity + contrast boost
float intensity = (float)light.getIntensityAt(point) * 1.1f;
return ColorUtil.adjustContrast(
    ColorUtil.multiply(finalColor, intensity),
    1.2f
);
}

// --- PBRCapableMaterial Implementation ---
@Override public Color getAlbedo() { return albedo; }
@Override public double getRoughness() { return roughness; }
@Override public double getMetalness() { return metalness; }
@Override public MaterialType getMaterialType() { return
MaterialType.METAL; }
@Override public double getReflectivity() { return 0.9 * metalness; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.0; }

}

// =====
// File: /net/elenamurat/material/pbr/CeramicTilePBRMaterial.java
// =====

package net.elenamurat.material.pbr;

```

```
import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CeramicTilePBRMaterial implements PBRCapableMaterial {
    protected final Color tileColor;
    private final Color groutColor;
    private final double tileSize;
    private final double groutWidth;
    private final double tileRoughness;
    private final double groutRoughness;
    private final double tileSpecular;
    private final double groutSpecular;

    // Advanced PBR
    private final double fresnelIntensity;
    private final double normalMicroFacet;
    private final double reflectionSharpness;
    private final double energyConservation;

    // Default values (glossy ceramic)
    public static final Color DEFAULT_TILE_COLOR = new Color(245,
245, 255); // Ice white
    public static final Color DEFAULT_GROUT_COLOR = new Color(70,
70, 80); // Blackish grout
    public static final double DEFAULT_TILE_SIZE = 0.6;
    public static final double DEFAULT_GROUT_WIDTH = 0.03;
    public static final double DEFAULT_TILE_ROUGHNESS = 0.1; //
Very glossy
    public static final double DEFAULT_GROUT_ROUGHNESS = 0.6; //
Semi-matte grout
    public static final double DEFAULT_TILE_SPECULAR = 0.5;
    public static final double DEFAULT_GROUT_SPECULAR = 0.3;

    // Default advanced PBR values
    public static final double DEFAULT_FRESNEL_INTENSITY = 0.9;
```

```
    public static final double DEFAULT_NORMAL_MICRO_FACET =  
0.02;  
    public static final double DEFAULT_REFLECTION_SHARPNESS =  
0.95;  
    public static final double DEFAULT_ENERGY_CONSERVATION =  
1.0;
```

```
public CeramicTilePBRMaterial() {  
    this(DEFAULT_TILE_COLOR, DEFAULT_GROUT_COLOR,  
        DEFAULT_TILE_SIZE, DEFAULT_GROUT_WIDTH,  
        DEFAULT_TILE_ROUGHNESS,  
        DEFAULT_GROUT_ROUGHNESS,  
        DEFAULT_TILE_SPECULAR,  
        DEFAULT_GROUT_SPECULAR,  
        DEFAULT_FRESNEL_INTENSITY,  
        DEFAULT_NORMAL_MICRO_FACET,  
        DEFAULT_REFLECTION_SHARPNESS,  
        DEFAULT_ENERGY_CONSERVATION);  
}
```

```
public CeramicTilePBRMaterial(Color tileColor, Color groutColor,  
        double tileSize, double groutWidth,  
        double tileRoughness, double groutRoughness) {  
    this(tileColor, groutColor, tileSize, groutWidth,  
        tileRoughness, groutRoughness, DEFAULT_TILE_SPECULAR,  
        DEFAULT_GROUT_SPECULAR,  
        DEFAULT_FRESNEL_INTENSITY,  
        DEFAULT_NORMAL_MICRO_FACET,  
        DEFAULT_REFLECTION_SHARPNESS,  
        DEFAULT_ENERGY_CONSERVATION);  
}
```

```
public CeramicTilePBRMaterial(Color tileColor, Color groutColor,  
        double tileSize, double groutWidth,  
        double tileRoughness, double groutRoughness,  
        double tileSpecular, double groutSpecular) {  
    this(tileColor, groutColor, tileSize, groutWidth,  
        tileRoughness, groutRoughness, tileSpecular, groutSpecular,  
        DEFAULT_FRESNEL_INTENSITY,
```

```

DEFAULT_NORMAL_MICRO_FACET,
    DEFAULT_REFLECTION_SHARPNESS,
DEFAULT_ENERGY_CONSERVATION);
}

// Full constructor
public CeramicTilePBRMaterial(Color tileColor, Color groutColor,
        double tileSize, double groutWidth,
        double tileRoughness, double groutRoughness,
        double tileSpecular, double groutSpecular,
        double fresnelIntensity, double normalMicroFacet,
        double reflectionSharpness, double
energyConservation) {
    this.tileColor = ColorUtil.adjustSaturation(tileColor, 1.2f);
    this.groutColor = groutColor;
    this.tileSize = Math.max(0.1, tileSize);
    this.groutWidth = Math.max(0.01, Math.min(0.1, groutWidth));
    this.tileRoughness = clamp(tileRoughness, 0.01, 1.0);
    this.groutRoughness = clamp(groutRoughness, 0.01, 1.0);
    this.tileSpecular = clamp(tileSpecular, 0.0, 1.0);
    this.groutSpecular = clamp(groutSpecular, 0.0, 1.0);

    // Advanced
    this.fresnelIntensity = clamp(fresnelIntensity, 0.0, 1.0);
    this.normalMicroFacet = clamp(normalMicroFacet, 0.0, 0.1);
    this.reflectionSharpness = clamp(reflectionSharpness, 0.5, 1.0);
    this.energyConservation = clamp(energyConservation, 0.8, 1.0);
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Tile pattern detection with proper UV mapping
    double u = ((point.x % tileSize) + tileSize) % tileSize / tileSize;
    double v = ((point.z % tileSize) + tileSize) % tileSize / tileSize;

    boolean isGroutLocal = (u < groutWidth/tileSize || u > 1.0 -
groutWidth/tileSize ||
                           v < groutWidth/tileSize || v > 1.0 -

```

```

groutWidth/tileSize);

// 2. Select base color and material properties
Color baseColor = isGroutLocal ? groutColor : tileColor;
double roughness = isGroutLocal ? groutRoughness : tileRoughness;
double specularIntensity = isGroutLocal ? groutSpecular :
tileSpecular;

// 3. Light direction calculations
Vector3 lightDir = light.getDirectionTo(point).normalize();
Vector3 viewDir = new Vector3(point, viewerPos).normalize();

// 4. Fresnel
double cosTheta = Math.max(0.001, normal.dot(viewDir));
double fresnel = Math.pow(1.0 - cosTheta, 5.0);
fresnel = fresnelIntensity * (0.04 + 0.96 * fresnel);

// 5. Mikro roughness
Vector3 perturbedNormal = applyMicroFacet(normal, point);

// 6. Diffuse component (Lambertian) - Energy conservation
double NdotL = Math.max(0.0, perturbedNormal.dot(lightDir));
Color diffuseColor = ColorUtil.multiply(baseColor, (float)(NdotL *
energyConservation));

// 7. Specular component - Yeni özelliklerle geliştirilmiş
Vector3 reflectDir = reflect(lightDir.negate(), perturbedNormal);
double RdotV = Math.max(0.0, reflectDir.dot(viewDir));
double specularPower = (1.0 - roughness * reflectionSharpness) *
256.0 + 1.0;
double specular = specularIntensity * fresnel * Math.pow(RdotV,
specularPower);

// 8. Ambient component
double ambientIntensity = isGroutLocal ? 0.3 : 0.2;
Color ambientColor = ColorUtil.multiply(baseColor,
(float)ambientIntensity);

// 9. Combine all components

```

```

Color finalColor = ColorUtil.add(ambientColor, diffuseColor);
finalColor = ColorUtil.add(finalColor,
    ColorUtil.multiply(Color.WHITE, (float)specular));

// 10. Apply contrast and gamma correction
finalColor = ColorUtil.adjustContrast(finalColor, 1.1f);
return ColorUtil.gammaCorrect(finalColor, 0.8f);
}

/**
 * Mikro roughness
 */
private Vector3 applyMicroFacet(Vector3 normal, Point3 point) {
    if (normalMicroFacet <= 0.0) {
        return normal;
    }

    // Simple normal perturbation
    double noise = Math.sin(point.x * 10.0) * Math.cos(point.z * 10.0) *
0.01;
    Vector3 perturbation = new Vector3(
        noise * normalMicroFacet,
        (1.0 - Math.abs(noise)) * normalMicroFacet,
        noise * normalMicroFacet * 0.5
    );

    return normal.add(perturbation).normalize();
}

private Vector3 reflect(Vector3 incident, Vector3 normal) {
    return incident.subtract(normal.multiply(2.0 * incident.dot(normal)));
}

// PBR Properties
@Override
public Color getAlbedo() {
    return ColorUtil.blendColors(tileColor, groutColor, 0.7f);
}

```

```
@Override
public double getRoughness() {
    return (tileRoughness + groutRoughness) / 2.0;
}

@Override
public double getMetalness() {
    return 0.0;
}

@Override
public MaterialType getMaterialType() {
    return MaterialType.DIELECTRIC;
}

@Override
public double getReflectivity() {
    return 0.4 * fresnelIntensity; // Fresnel adding
}

@Override
public double getIndexOfRefraction() {
    return 1.52;
}

@Override
public double getTransparency() {
    return 0.0;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    // Optional: Implement if needed for texture mapping
}

// Helper methods
private double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}
```

```
// Getter metodlari
public Color getTileColor() { return tileColor; }
public Color getGroutColor() { return groutColor; }
public double getTileRoughness() { return tileRoughness; }
public double getGroutRoughness() { return groutRoughness; }
public double getTileSize() { return tileSize; }
public double getGroutWidth() { return groutWidth; }
public double getTileSpecular() { return tileSpecular; }
public double getGroutSpecular() { return groutSpecular; }

// Getters advanced properties
public double getFresnelIntensity() { return fresnelIntensity; }
public double getNormalMicroFacet() { return normalMicroFacet; }
public double getReflectionSharpness() { return reflectionSharpness; }
public double getEnergyConservation() { return energyConservation; }

}
```

```
// =====
// File: /net/elenamurat/material/pbr/PBRCapableMaterial.java
// =====
```

```
package net.elena.murat.material.pbr;

import java.awt.Color;

import net.elena.murat.light.Light;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.material.Material; // Temel arayüzü extend ediyor

/**
 * PBRCapableMaterial is an interface for materials that support
 * Physically Based Rendering.
 * It extends the basic Material interface and adds PBR-specific properties.
 */
public interface PBRCapableMaterial extends Material {
```

```
/**  
 * Returns the base color (albedo) of the material.  
 * @return The albedo color.  
 */  
Color getAlbedo();  
  
/**  
 * Returns the surface roughness [0.0 (smooth) to 1.0 (rough)].  
 * @return The roughness value.  
 */  
double getRoughness();  
  
/**  
 * Returns the metalness factor [0.0 (dielectric) to 1.0 (metal)].  
 * @return The metalness value.  
 */  
double getMetalness();  
  
/**  
 * Returns the type of PBR material (e.g., METAL, DIELECTRIC).  
 * @return The material type.  
 */  
MaterialType getMaterialType();  
}
```

```
// =====  
// File: /net/lena/murat/material/Material.java  
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```
//custom  
import net.elena.murat.light.Light;  
import net.elena.murat.math.*;
```

```
/**
```

```
* Material interface defines the contract for all materials in the ray tracer.  
* It now includes methods for color calculation at a point,  
* as well as properties for reflectivity, index of refraction, and  
transparency.
```

```
*/
```

```
public interface Material {
```

```
/**
```

```
* Calculates the final color at a given point on the surface, considering  
* the material's properties and the light source.
```

```
* @param point The point in 3D space (world coordinates) where the  
light hits.
```

```
* @param normal The normal vector at the point (world coordinates).
```

```
* @param light The single light source affecting this point.
```

```
* @param viewerPos The position of the viewer/camera.
```

```
* @return The color contribution from this specific light for the point.
```

```
*/
```

```
Color getColorAt(Point3 point, Vector3 normal, Light light, Point3  
viewerPos);
```

```
/**
```

```
* Returns the reflectivity coefficient of the material.
```

```
* This value determines how much light is reflected by the surface (0.0  
for no reflection, 1.0 for full reflection).
```

```
* @return The reflectivity value (0.0-1.0).
```

```
*/
```

```
double getReflectivity();
```

```
/**
```

```
* Returns the index of refraction (IOR) of the material.
```

```
* This value is used for calculating refraction (transparency).
```

```
* @return The index of refraction (typically 1.0 for air, less than 1.0 for  
denser materials).
```

```
*/
```

```
double getIndexOfRefraction();
```

```
/**
```

```
* Returns the transparency coefficient of the material.
```

```
* This value determines how much light passes through the surface (0.0  
for opaque, 1.0 for fully transparent).
```



```

// Phong lighting model parameters
private final double ambientCoefficient;
private final double diffuseCoefficient;
private final double specularCoefficient;
private final double shininess;
private final double reflectivity;
private final double ior; // Index of Refraction
private final double transparency;

private final Color specularColor = Color.WHITE; // Default specular
color for highlights

/***
 * Full constructor for CircleTextureMaterial.
 * @param solidColor The background color of the material.
 * @param patternColor The color of the dots.
 * @param patternSize Controls the density/spacing of the circles.
Smaller value means more circles.
 * @param ambientCoefficient Ambient light contribution.
 * @param diffuseCoefficient Diffuse light contribution.
 * @param specularCoefficient Specular light contribution.
 * @param shininess Shininess for specular highlights.
 * @param reflectivity Material's reflectivity (0.0 to 1.0).
 * @param ior Index of refraction for transparent materials.
 * @param transparency Material's transparency (0.0 to 1.0).
 * @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.
*/
public CircleTextureMaterial(Color solidColor, Color patternColor,
double patternSize,
    double ambientCoefficient, double diffuseCoefficient,
    double specularCoefficient, double shininess,
    double reflectivity, double ior, double transparency,
    Matrix4 objectInverseTransform) { // Added objectInverseTransform
this.solidColor = solidColor;
this.patternColor = patternColor;
this.patternSize = patternSize;
this.circleRadius = 0.35; // Corrected: Fixed radius for the circle within
a [0,1) cell.

```

```

// This value can be adjusted to make circles larger/smaller.
this.objectInverseTransform = objectInverseTransform; // Store the
inverse transform

this.ambientCoefficient = ambientCoefficient;
this.diffuseCoefficient = diffuseCoefficient;
this.specularCoefficient = specularCoefficient;
this.shininess = shininess;
this.reflectivity = reflectivity;
this.ior = ior;
this.transparency = transparency;
}

/***
 * Simplified constructor with default Phong-like coefficients.
 * @param solidColor The background color of the material.
 * @param patternColor The color of the dots.
 * @param patternSize Controls the density/spacing of the circles.
Smaller value means more circles.
 * @param objectInverseTransform The full inverse transformation
matrix of the object.
 */
public CircleTextureMaterial(Color solidColor, Color patternColor,
double patternSize, Matrix4 objectInverseTransform) { // Added
objectInverseTransform
    this(solidColor, patternColor, patternSize,
        0.1, // ambientCoefficient
        0.8, // diffuseCoefficient
        0.1, // specularCoefficient
        10.0, // shininess
        0.0, // reflectivity
        1.0, // indexOfRefraction
        0.0, // transparency
        objectInverseTransform // Pass the inverse transform
    );
}

@Override
public void setObjectTransform(Matrix4 tm) {

```

```

if (tm == null) tm = new Matrix4 ();
this.objectInverseTransform = tm;
}

/***
* Calculates the base pattern color (solid or circle) at a given local point.
* This method now handles the pattern repetition based on patternSize.
*
* @param localPoint The 3D point in the object's local coordinates.
* @param localNormal The surface normal at that point in local
coordinates.
* @return The pattern color (solidColor or patternColor).
*/
private Color getPatternColor(Point3 localPoint, Vector3 localNormal)
{ // Changed parameters to Point3 and Vector3
    // Use normal-based planar mapping to get UV coordinates
    double u, v;
    double absNx = Math.abs(localNormal.x);
    double absNy = Math.abs(localNormal.y);
    double absNz = Math.abs(localNormal.z);

    // Project the 3D local point onto a 2D plane based on the dominant
    local normal axis.
    // Then scale by 'patternSize' to control repetition.
    if (absNx > absNy && absNx > absNz) { // Normal is mostly X-axis
        (local Y-Z plane)
        u = localPoint.y * this.patternSize;
        v = localPoint.z * this.patternSize;
    } else if (absNy > absNx && absNy > absNz) { // Normal is mostly Y-
        axis (local X-Z plane)
        u = localPoint.x * this.patternSize;
        v = localPoint.z * this.patternSize;
    } else { // Normal is mostly Z-axis (local X-Y plane) or equally
        dominant
        u = localPoint.x * this.patternSize;
        v = localPoint.y * this.patternSize;
    }

    // Normalize UV values to [0, 1) range within a single pattern cell
}

```

```

    // We use floor to get the integer part of the scaled coordinate, and
    subtract it
    // to get the fractional part, which represents the position within the
    current cell.
    double normalizedU = u - Math.floor(u + Ray.EPSILON);
    double normalizedV = v - Math.floor(v + Ray.EPSILON);

    // Calculate distance from the center of the current UV cell (0.5, 0.5)
    double dx = normalizedU - 0.5;
    double dy = normalizedV - 0.5;
    double distance = Math.sqrt(dx * dx + dy * dy);

    // If the point is within the circleRadius, return patternColor, otherwise
    solidColor.
    if (distance <= circleRadius) {
        return patternColor;
    } else {
        return solidColor;
    }

    /**
     * Returns the final shaded color at a given 3D point on the surface,
     * applying the circular pattern and a Phong-like lighting model.
     * Uses normal-based planar UV mapping for better generalization across
     shapes.
     *
     * @param worldPoint The 3D point on the surface in world coordinates.
     * @param worldNormal The surface normal at that point in world
     coordinates (should be normalized).
     * @param light The light source.
     * @param viewerPos The position of the viewer/camera.
     * @return The final shaded color.
     */
    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
        // Check if inverse transform is valid before proceeding
        if (objectInverseTransform == null) {

```

```

    System.err.println("Error: CircleTextureMaterial's inverse transform is
null. Returning black.");
    return Color.BLACK;
}

// 1. Transform the intersection point from world space into the object's
local space
Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);

// 2. Transform the world normal to local space to determine the local
face orientation
Vector3 localNormal =
objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();
// Check if the transformed normal is valid
if (localNormal == null) {
    System.err.println("Error: CircleTextureMaterial's normal transform
matrix is null or invalid. Returning black.");
    return Color.BLACK;
}

// Get the base pattern color from the circular procedural pattern.
// Pass localPoint and localNormal to getPatternColor for UV
calculation
Color patternBaseColor = getPatternColor(localPoint, localNormal);

// --- Lighting Calculation (Ambient + Diffuse + Specular Phong Model)
---
// This part is consistent with your other material classes.

Color lightColor = light.getColor();

// Ambient component
// Ensure calculations are done with doubles and then clamped
double rAmbient = patternBaseColor.getRed() / 255.0 *
ambientCoefficient * lightColor.getRed() / 255.0;
double gAmbient = patternBaseColor.getGreen() / 255.0 *
ambientCoefficient * lightColor.getGreen() / 255.0;

```

```

        double bAmbient = patternBaseColor.getBlue() / 255.0 *
ambientCoefficient * lightColor.getBlue() / 255.0;

        if (light instanceof ElenaMuratAmbientLight) {
            return new Color(
                (float)Math.min(1.0, rAmbient),
                (float)Math.min(1.0, gAmbient),
                (float)Math.min(1.0, bAmbient)
            );
        }

Vector3 lightDirection;
if (light instanceof MuratPointLight) {
    lightDirection = ((MuratPointLight)
light).getPosition().subtract(worldPoint).normalize();
} else if (light instanceof ElenaDirectionalLight) {
    lightDirection = ((ElenaDirectionalLight)
light).getDirection().negate().normalize();
} else if (light instanceof PulsatingPointLight) {
    lightDirection = ((PulsatingPointLight)
light).getPosition().subtract(worldPoint).normalize();
} else if (light instanceof BioluminescentLight) {
    lightDirection = ((BioluminescentLight)
light).getDirectionAt(worldPoint).normalize();
} else if (light instanceof BlackHoleLight) {
    lightDirection = ((BlackHoleLight)
light).getDirectionAt(worldPoint).normalize();
} else if (light instanceof FractalLight) {
    lightDirection = ((FractalLight)
light).getDirectionAt(worldPoint).normalize();
} else {
    System.err.println("Warning: Unknown or unsupported light type for
CircleTextureMaterial shading: " + light.getClass().getName());
    return Color.BLACK;
}

// Diffuse component
double NdotL = Math.max(0, worldNormal.dot(lightDirection));
double rDiffuse = patternBaseColor.getRed() / 255.0 *

```

```

diffuseCoefficient * lightColor.getRed() / 255.0 * NdotL;
    double gDiffuse = patternBaseColor.getGreen() / 255.0 *
diffuseCoefficient * lightColor.getGreen() / 255.0 * NdotL;
    double bDiffuse = patternBaseColor.getBlue() / 255.0 *
diffuseCoefficient * lightColor.getBlue() / 255.0 * NdotL;

// Specular component
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectionVector =
lightDirection.negate().reflect(worldNormal); // Use negate() before
reflect() for correct reflection vector
double RdotV = Math.max(0, reflectionVector.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

double rSpecular = specularColor.getRed() / 255.0 * specularCoefficient
* lightColor.getRed() / 255.0 * specFactor;
    double gSpecular = specularColor.getGreen() / 255.0 *
specularCoefficient * lightColor.getGreen() / 255.0 * specFactor;
    double bSpecular = specularColor.getBlue() / 255.0 *
specularCoefficient * lightColor.getBlue() / 255.0 * specFactor;

// Sum up all components for this light source
double finalR = rAmbient + rDiffuse + rSpecular;
double finalG = gAmbient + gDiffuse + gSpecular;
double finalB = bAmbient + bDiffuse + bSpecular;

return new Color(
    (float)clamp01(finalR),
    (float)clamp01(finalG),
    (float)clamp01(finalB)
);
}

/**
 * Returns the reflectivity coefficient of the material.
 * @return The reflectivity value (0.0-1.0).
 */
@Override
public double getReflectivity() { return reflectivity; }

```

```

/**
 * Returns the index of refraction (IOR) of the material.
 * @return The index of refraction.
 */
@Override
public double getIndexOfRefraction() { return ior; }

/**
 * Returns the transparency coefficient of the material.
 * @return The transparency value (0.0-1.0).
 */
@Override
public double getTransparency() { return transparency; }

/**
 * Helper method to clamp a double value between 0.0 and 1.0.
 * @param val The value to clamp.
 * @return A value between 0.0 and 1.0.
 */
private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}

/**
 * Helper method to get the normalized light direction vector from a light
source to a point.
 * @param light The light source.
 * @param worldPoint The point in world coordinates.
 * @return Normalized light direction vector, or null if light type is
unsupported.
 */
private Vector3 getLightDirection(Light light, Point3 worldPoint) {
    if (light instanceof MuratPointLight) {
        return
((MuratPointLight)light).getPosition().subtract(worldPoint).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return
((ElenaDirectionalLight)light).getDirection().negate().normalize();
    }
}

```

```

        } else if (light instanceof PulsatingPointLight) {
            return
((PulsatingPointLight)light).getPosition().subtract(worldPoint).normalize()
;
        } else if (light instanceof SpotLight) {
            return ((SpotLight)light).getDirectionAt(worldPoint).normalize();
        } else if (light instanceof BioluminescentLight) {
            return
((BioluminescentLight)light).getDirectionAt(worldPoint).normalize();
        } else if (light instanceof BlackHoleLight) {
            return
((BlackHoleLight)light).getDirectionAt(worldPoint).normalize();
        } else if (light instanceof FractalLight) {
            return ((FractalLight)light).getDirectionAt(worldPoint).normalize();
        } else {
            System.err.println("Warning: Unknown light type in
PlaneFaceElenaTextureMaterial");
            return new Vector3(0, 1, 0); // Varsayılan yön
        }
    }

}

```

```

// =====
// File: /net/elenamurat/material/EmeraldMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

public class EmeraldMaterial implements Material {

```

```

private Color baseColor;
private double density;
private double reflectivity;
private Matrix4 objectTransform;

public EmeraldMaterial() {
    this.baseColor = new Color(20, 220, 60);
    this.density = 0.8;
    this.reflectivity = 0.12;
}

public EmeraldMaterial(Color baseColor, double density, double
reflectivity) {
    this.baseColor = baseColor;
    this.density = Math.max(0, Math.min(1, density));
    this.reflectivity = Math.max(0, Math.min(1, reflectivity));
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Color lightColor = light.getColor();

    double red = (lightColor.getRed() * baseColor.getRed() / 255.0) *
density * 0.3;
    double green = (lightColor.getGreen() * baseColor.getGreen() / 255.0) *
density;
    double blue = (lightColor.getBlue() * baseColor.getBlue() / 255.0) *
density * 0.4;

    int r = ColorUtil.clampColorValue((int) red);
    int g = ColorUtil.clampColorValue((int) green);
    int b = ColorUtil.clampColorValue((int) blue);

    return new Color(r, g, b);
}

@Override

```

```
public void setObjectTransform(Matrix4 tm) {  
    this.objectTransform = tm;  
}
```

```
@Override  
public double getTransparency() {  
    return 0.75;  
}
```

```
@Override  
public double getIndexOfRefraction() {  
    return 1.58;  
}
```

```
@Override  
public double getReflectivity() {  
    return this.reflectivity;  
}
```

```
public Color getBaseColor() {  
    return baseColor;  
}
```

```
public void setBaseColor(Color baseColor) {  
    this.baseColor = baseColor;  
}
```

```
public double getDensity() {  
    return density;  
}
```

```
public void setDensity(double density) {  
    this.density = Math.max(0, Math.min(1, density));  
}
```

```
public double getReflectivityValue() {  
    return reflectivity;  
}
```

```
public void setReflectivity(double reflectivity) {
    this.reflectivity = Math.max(0, Math.min(1, reflectivity));
}

}

// =====
// File: /net/elenamurat/material/DynamicGlassMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

/**
 * DynamicGlassMaterial - Glass material with optional per-pixel dynamic
parameters.
 *
 * If per-pixel IOR, transparency or reflectivity are constant, set methods
are not called inside getColorAt.
 * If per-pixel variation is needed, override getColorAt to update
parameters via set methods.
 */
public class DynamicGlassMaterial implements Material {

    // Base physical properties (default values)
    private double baseIOR = 1.5;
    private double baseReflectivity = 0.04;
    private double baseTransparency = 0.92;
    private Color tintColor = new Color(200, 220, 240);
    private double roughness = 0.01;
    private double density = 0.1;
    private double chromaticAberrationStrength = 0.001;
```

```

// Flag to enable per-pixel dynamic parameter update
private boolean dynamicParametersEnabled = false;

public DynamicGlassMaterial() {
}

public DynamicGlassMaterial(Color tintColor,
    double reflectivity,
    double transparency,
    double roughness,
    double density,
    double chromaticAberrationStrength) {
    this.tintColor = tintColor;
    this.baseReflectivity = clamp(reflectivity, 0.0, 1.0);
    this.baseTransparency = clamp(transparency, 0.0, 1.0);
    this.roughness = clamp(roughness, 0.0, 1.0);
    this.density = clamp(density, 0.0, 5.0);
    this.chromaticAberrationStrength =
        clamp(chromaticAberrationStrength, 0.0, 0.1);
}

/**
 * Enable or disable per-pixel dynamic parameter update.
 * If enabled, getColorAt will update IOR, transparency, reflectivity per
pixel.
 */
public void setDynamicParametersEnabled(boolean enabled) {
    this.dynamicParametersEnabled = enabled;
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Vector3 viewDir = viewerPoint.subtract(point).normalize();

    if (dynamicParametersEnabled) {
        // Example dynamic update: vary parameters based on angle between
normal and viewDir
        double angleFactor = Math.abs(normal.dot(viewDir));
    }
}

```

```

        double dynamicIOR = 1.4 + 0.2 * angleFactor; // 1.4 - 1.6
        double dynamicTransparency = 0.8 + 0.2 * (1.0 - angleFactor); // 0.8 -
1.0
        double dynamicReflectivity = 0.05 + 0.1 * angleFactor; // 0.05 - 0.15

        setBaseIOR(dynamicIOR);
        setBaseTransparency(dynamicTransparency);
        setBaseReflectivity(dynamicReflectivity);
    }

    // Chromatic aberration per channel
    double iorR = clamp(baseIOR + chromaticAberrationStrength, 1.0, 3.0);
    double iorG = clamp(baseIOR, 1.0, 3.0);
    double iorB = clamp(baseIOR - chromaticAberrationStrength, 1.0, 3.0);

    Color colorR = calculateRefractReflectColor(point, normal, light,
viewDir, iorR);
    Color colorG = calculateRefractReflectColor(point, normal, light,
viewDir, iorG);
    Color colorB = calculateRefractReflectColor(point, normal, light,
viewDir, iorB);

    int r = clamp(colorR.getRed(), 0, 255);
    int g = clamp(colorG.getGreen(), 0, 255);
    int b = clamp(colorB.getBlue(), 0, 255);

    return new Color(r, g, b);
}

private Color calculateRefractReflectColor(Point3 point, Vector3 normal,
Light light, Vector3 viewDir, double ior) {
    double cosTheta = Math.abs(viewDir.dot(normal));
    double r0 = Math.pow((1.0 - ior) / (1.0 + ior), 2);
    double fresnel = r0 + (1.0 - r0) * Math.pow(1.0 - cosTheta, 5.0);

    Vector3 perturbedNormal = perturbNormal(normal, roughness);

    Vector3 refractedDir = refract(viewDir, perturbedNormal, ior);
}

```

```

    Color refractedColor = calculateLightTransport(point, refractedDir,
light, baseTransparency * Math.exp(-density));

    Vector3 reflectedDir = reflect(viewDir, perturbedNormal);
    Color reflectedColor = calculateLightTransport(point, reflectedDir,
light, baseReflectivity);

    return ColorUtil.add(
        ColorUtil.multiplyColorFloat(refractedColor, (float)((1.0 - fresnel) *
baseTransparency)),
        ColorUtil.multiplyColorFloat(reflectedColor, (float)(fresnel *
baseReflectivity)))
    );
}

```

```
private Vector3 perturbNormal(Vector3 normal, double roughness) {
    if (roughness <= 0.0) return normal;
```

```
    Vector3 tangent = generateTangent(normal);
    Vector3 bitangent = normal.cross(tangent);
```

```
    double rand1 = (Math.random() - 0.5) * 2.0 * roughness;
    double rand2 = (Math.random() - 0.5) * 2.0 * roughness;
```

```
    Vector3 perturbed = normal
.add(tangent.multiply((float)rand1))
.add(bitangent.multiply((float)rand2))
.normalize();
```

```
    return perturbed;
}
```

```
private Vector3 generateTangent(Vector3 normal) {
    Vector3 up = Math.abs(normal.z) < 0.999 ? new Vector3(0, 0, 1) : new
Vector3(1, 0, 0);
    return normal.cross(up).normalize();
}
```

```
private Color calculateLightTransport(Point3 point, Vector3 direction,
```

```

Light light, double intensity) {
    Vector3 lightDir = light.getDirectionAt(point);
    if (lightDir.lengthSquared() < 1e-10) {
        return ColorUtil.multiplyColorFloat(tintColor, (float)intensity);
    }

    lightDir = lightDir.normalize();
    double lightFactor = Math.max(0.1, direction.dot(lightDir));
    double attenuatedIntensity = intensity * light.getIntensity();

    attenuatedIntensity *= Math.exp(-density);

    return ColorUtil.multiplyColorFloat(tintColor, (float)(lightFactor *
attenuatedIntensity));
}

private Vector3 refract(Vector3 incident, Vector3 normal, double ior) {
    double cosi = incident.dot(normal);
    double etai = 1.0, etat = ior;
    Vector3 n = normal;

    if (cosi < 0) {
        cosi = -cosi;
    } else {
        double temp = etai;
        etai = etat;
        etat = temp;
        n = normal.negate();
    }

    double eta = etai / etat;
    double k = 1.0 - eta * eta * (1.0 - cosi * cosi);

    if (k < 0) {
        return new Vector3(0, 0, 0);
    }

    return incident.multiply((float)eta)
.add(n.multiply((float)(eta * cosi - Math.sqrt(k)))))

}

```

```
    .normalize();
}

private Vector3 reflect(Vector3 incident, Vector3 normal) {
    double dot = incident.dot(normal);
    return incident.subtract(normal.multiply((float)(2.0 * dot))).normalize();
}

// Getters and setters

@Override
public double getIndexOfRefraction() {
    return baseIOR;
}

@Override
public double getReflectivity() {
    return baseReflectivity;
}

@Override
public double getTransparency() {
    return baseTransparency;
}

public void setBaseIOR(double ior) {
    this.baseIOR = clamp(ior, 1.0, 3.0);
}

public void setBaseReflectivity(double reflectivity) {
    this.baseReflectivity = clamp(reflectivity, 0.0, 1.0);
}

public void setBaseTransparency(double transparency) {
    this.baseTransparency = clamp(transparency, 0.0, 1.0);
}

public void setTintColor(Color tintColor) {
    if (tintColor != null) {
```

```
    this.tintColor = tintColor;
}
}

public void setRoughness(double roughness) {
    this.roughness = clamp(roughness, 0.0, 1.0);
}

public void setDensity(double density) {
    this.density = clamp(density, 0.0, 5.0);
}

public void setChromaticAberrationStrength(double strength) {
    this.chromaticAberrationStrength = clamp(strength, 0.0, 0.1);
}

private static double clamp(double val, double min, double max) {
    return Math.max(min, Math.min(max, val));
}

private static int clamp(int val, int min, int max) {
    return Math.max(min, Math.min(max, val));
}

@Override
public void setObjectTransform(Matrix4 tm) {
    // No transform needed for this material
}

}

// =====
// File: /net/elenamurat/material/LambertMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
```

```

import net.elena.murat.math.*;
import net.elena.murat.light.*;

/**
 * LambertMaterial represents a matte, non-glossy surface that reflects
light
 * equally in all directions, using a simplified Phong-like model (ambient
and diffuse components).
 * It now fully implements the extended Material interface.
*/
public class LambertMaterial implements Material {
    private final Color diffuseColor; // This is the main (diffuse) color of the
material.

    // Material coefficients
    private final double ambientCoefficient;
    private final double diffuseCoefficient;

    // Default values for new interface methods, as this is a basic diffuse
material
    private final double reflectivity = 0.0;
    private final double ior = 1.0; // Index of Refraction for air/vacuum
    private final double transparency = 0.0;

    /**
     * Default constructor for LambertMaterial. Uses default ambient (0.1)
and diffuse (1.0) coefficients.
     * @param color The base color (diffuse color) of the material.
    */
    public LambertMaterial(Color color) {
        this.diffuseColor = color;
        this.ambientCoefficient = 0.1; // General ambient lighting contribution
        this.diffuseCoefficient = 1.0; // Use full diffuse lighting contribution
    }

    /**
     * Constructor to customize material coefficients.
     * @param color The base color of the material.
    */

```

```

    * @param ambientCoeff How much the material reacts to ambient light
(0.0-1.0).
    * @param diffuseCoeff How much the material reacts to diffuse light
(0.0-1.0).
    */
public LambertMaterial(Color color, double ambientCoeff, double
diffuseCoeff) {
    this.diffuseColor = color;
    this.ambientCoefficient = ambientCoeff;
    this.diffuseCoefficient = diffuseCoeff;
}

/***
 * Calculates the material color at a specific point, considering the
contribution of a single light source.
 * This method applies ambient and diffuse lighting components.
 *
 * @param point The intersection point in world coordinates.
 * @param normal The surface normal at the intersection point in world
coordinates.
 * @param light The single light source to be used in the calculation.
 * @param viewerPos The position of the viewer (camera) in world
coordinates (not directly used for Lambertian).
 * @return The calculated color contribution.
 */
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    Color lightColor = light.getColor();
    double attenuatedIntensity = 0.0; // Initialize for non-ambient lights

    // Ambient component
    int rAmbient = (int) (diffuseColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
    int gAmbient = (int) (diffuseColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
    int bAmbient = (int) (diffuseColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);
}

```

```

if (light instanceof ElenaMuratAmbientLight) {
    return new Color(
        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

```

```

Vector3 lightDirection;
if (light instanceof MuratPointLight) {
    MuratPointLight pLight = (MuratPointLight) light;
    lightDirection = pLight.getPosition().subtract(point).normalize();
    attenuatedIntensity = pLight.getAttenuatedIntensity(point);
} else if (light instanceof ElenaDirectionalLight) {
    ElenaDirectionalLight dLight = (ElenaDirectionalLight) light;
    lightDirection = dLight.getDirection().negate().normalize();
    attenuatedIntensity = dLight.getIntensity(); // Directional lights don't
attenuate with distance
} else if (light instanceof PulsatingPointLight) {
    PulsatingPointLight ppLight = (PulsatingPointLight) light;
    lightDirection = ppLight.getPosition().subtract(point).normalize();
    attenuatedIntensity = ppLight.getAttenuatedIntensity(point);
} else if (light instanceof SpotLight) {
    SpotLight sLight = (SpotLight) light;
    lightDirection = sLight.getDirectionAt(point);
    attenuatedIntensity = sLight.getAttenuatedIntensity(point);
} else if (light instanceof BioluminescentLight) {
    BioluminescentLight bLight = (BioluminescentLight) light;
    lightDirection = bLight.getDirectionAt(point);
    attenuatedIntensity = bLight.getAttenuatedIntensity(point);
} else if (light instanceof BlackHoleLight) {
    BlackHoleLight bhLight = (BlackHoleLight) light;
    lightDirection = bhLight.getDirectionAt(point);
    attenuatedIntensity = bhLight.getAttenuatedIntensity(point);
} else if (light instanceof FractalLight) {
    FractalLight fLight = (FractalLight) light;
    lightDirection = fLight.getDirectionAt(point);
    attenuatedIntensity = fLight.getAttenuatedIntensity(point);
} else {

```

```

    System.err.println("Warning: Unknown or unsupported light type for
Lambertian shading: " + light.getClass().getName());
    return Color.BLACK;
}

// Diffuse component
double NdotL = Math.max(0, normal.dot(lightDirection));
int rDiffuse = (int) (diffuseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int) (diffuseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int) (diffuseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

// Sum up ambient and diffuse components for this light source
// Note: Ambient component is added only once per pixel in RayTracer's
shade method,
// so here we only return the diffuse contribution for non-ambient lights.
// The rAmbient, gAmbient, bAmbient are calculated but not added to
finalR/G/B here
// if it's not an AmbientLight.
int finalR = Math.min(255, rDiffuse);
int finalG = Math.min(255, gDiffuse);
int finalB = Math.min(255, bDiffuse);

return new Color(finalR, finalG, finalB);
}

/**
 * Returns the reflectivity coefficient. For a Lambertian material, this is
typically 0.0.
 * @return 0.0
 */
@Override
public double getReflectivity() {
    return reflectivity;
}

/**

```

```
* Returns the index of refraction. For an opaque Lambertian material,  
this is typically 1.0.  
 * @return 1.0  
 */  
@Override  
public double getIndexOfRefraction() {  
    return ior;  
}  
  
/**  
 * Returns the transparency coefficient. For an opaque Lambertian  
material, this is typically 0.0.  
 * @return 0.0  
 */  
@Override  
public double getTransparency() {  
    return transparency;  
}  
  
@Override  
public void setObjectTransform(Matrix4 tm) {  
}  
}
```

```
// ======  
// File: /net/elenamurat/material/GradientImageTextMaterial.java  
// ======
```

```
package net.elena.murat.material;
```

```
import java.awt.*;  
import java.awt.geom.*;  
import java.awt.image.*;  
import java.util.Random;  
import net.elena.murat.light.*;  
import net.elena.murat.math.*;
```

```
public class GradientImageTextMaterial implements Material {  
    private final Color bgStartColor;  
    private final Color bgEndColor;  
    private final Color textStartColor;  
    private final Color textEndColor;  
    private final String text;  
    private final Font font;  
    private final StripeDirection direction;  
    private final double reflectivity;  
    private final double ior;  
    private final double transparency;  
    private Matrix4 objectInverseTransform;  
    private final int xOffset;  
    private final int yOffset;  
    private final int imgOffsetX;  
    private final int imgOffsetY;  
    private final BufferedImage bgImage;  
    private final float bgAlpha;  
    private final float textAlpha;  
    private BufferedImage texture;  
    private final boolean isWrap;  
  
    private Random random = new Random();  
  
    // Main constructor with all parameters  
    public GradientImageTextMaterial(Color bgStart, Color bgEnd,  
        Color textStart, Color textEnd,  
        BufferedImage bgImage, float bgAlpha, float textAlpha,  
        String text, Font font, StripeDirection direction,  
        double reflectivity, double ior, double transparency,  
        Matrix4 objectInverseTransform,  
        int xOffset, int yOffset,  
        int imgOffsetX, int imgOffsetY, boolean isWrap) {  
        this.bgStartColor = bgStart;  
        this.bgEndColor = bgEnd;  
        this.textStartColor = textStart;  
        this.textEndColor = textEnd;  
        this.bgImage = bgImage;  
        this.bgAlpha = bgAlpha;
```

```
this.textAlpha = textAlpha;
this.text = text;
this.font = font;
this.direction = direction;
this.reflectivity = Math.min(1.0, Math.max(0.0, reflectivity));
this.ior = Math.max(1.0, ior);
this.transparency = Math.min(1.0, Math.max(0.0, transparency));
this.objectInverseTransform = objectInverseTransform;
this.xOffset = xOffset;
this.yOffset = yOffset;
this.imgOffsetX = imgOffsetX;
this.imgOffsetY = imgOffsetY;
this.isWrap = isWrap;

this.texture = createCompositeTexture();
}

// Simplified constructor with default parameters
public GradientImageTextMaterial(BufferedImage bmg, String text) {
    this(bmg, text, 0, 0);
}

// Constructor with text and position offsets
public GradientImageTextMaterial(BufferedImage bimgo, String text, int
xOffset, int yOffset) {
    this(
        generateRandomColor(),
        generateRandomColor(),
        Color.WHITE,
        Color.BLACK,
        bimgo,
        1F,
        1F,
        text,
        new Font("Arial", Font.BOLD, 72),
        StripeDirection.RANDOM,
        0.3,
        1.0,
        0.1,
```

```

        new Matrix4 (),
        xOffset,
        yOffset,
        0,
        0,
        false
    );
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

private static Color generateRandomColor() {
    Random rand = new Random();
    return new Color(rand.nextFloat(), rand.nextFloat(), rand.nextFloat());
}

/**
 * Creates a texture with cyclic (repeating) gradients for strong visual
impact.
 * Uses GradientPaint with isCyclic=true to create wave-like color
transitions
 * that remain visible even after spherical mapping and on low-resolution
renders.
 *
 * @return A BufferedImage with repeating gradient patterns suitable for
3D materials.
 */
private BufferedImage createCompositeTexture() {
    final int size = 1024;
    BufferedImage texture = new BufferedImage(size, size,
BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = texture.createGraphics();

    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
}

```

```

g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

// === 1. BACKGROUND: CYCLIC GRADIENT ===
StripeDirection bgDir = (direction == StripeDirection.RANDOM)
? StripeDirection.values()[random.nextInt(3)]
: direction;

Point2D bgStart = new Point2D.Float(0, 0);
Point2D bgEnd;

// Define a short vector to make gradient repeat frequently
switch (bgDir) {
    case HORIZONTAL:
        bgEnd = new Point2D.Float((float)(size * 0.2), 0.0f); // Repeat every
20% horizontally
        break;
    case VERTICAL:
        bgEnd = new Point2D.Float(0.0f, (float)(size * 0.2)); // Repeat every
20% vertically
        break;
    case DIAGONAL:
        bgEnd = new Point2D.Float((float)(size * 0.15), (float)(size *
0.15)); // Short diagonal
        break;
    default:
        bgEnd = new Point2D.Float((float)(size * 0.2), 0.0f);
}
}

// isCyclic = true -> gradient repeats infinitely across the texture
GradientPaint bgGradient = new GradientPaint(
    bgStart, bgStartColor,
    bgEnd, bgEndColor,
    true
);
g2d.setPaint(bgGradient);
g2d.fillRect(0, 0, size, size);

if (bgImage != null) {

```

```

Composite acomp=AlphaComposite.getInstance (3, bgAlpha);
g2d.setComposite (acomp);

if (isWrap) {
    g2d.drawImage (bgImage, imgOffsetX, imgOffsetY, size, size, null);
} else {
    g2d.drawImage (bgImage, imgOffsetX, imgOffsetY, null);
}
}

// === 2. TEXT: CYCLIC GRADIENT MASK ===
if (text != null && !text.isEmpty()) {
    // Fix: Do not assign to 'font' if it's final
    // Use a local font or assume 'font' is already set
    Font renderFont = font != null ? font : new Font("Arial", Font.BOLD,
size / 6);
    g2d.setFont(renderFont);

    FontMetrics fm = g2d.getFontMetrics();
    int textWidth = fm.stringWidth(text);
    int textHeight = fm.getHeight();
    int ascent = fm.getAscent();

    int x = (size - textWidth) / 2 + xOffset;
    int y = (size - textHeight) / 2 + ascent + yOffset;

    x = Math.max(0, Math.min(size - textWidth, x));
    y = Math.max(ascent, Math.min(size - fm.getDescent(), y));

    // Define cyclic gradient direction for the text
    Point2D textStart = new Point2D.Float((float)x, (float)y);

    Point2D textEnd;

    switch (direction) {
        case HORIZONTAL:
            textEnd = new Point2D.Float(x + textWidth, y);
            break;
        case VERTICAL:

```

```

    textEnd = new Point2D.Float(x, y + textHeight);
    break;
  case DIAGONAL:
    textEnd = new Point2D.Float(x + textWidth, y + textHeight);
    break;

  default:
    textEnd = new Point2D.Float(x + textWidth, y);
}

// Text gradient is also cyclic!
GradientPaint textGradient = new GradientPaint(
  textStart, textStartColor,
  textEnd, textEndColor,
  true // <<< REPEATING TEXT GRADIENT
);
g2d.setPaint(textGradient);

Composite acomp=AlphaComposite.getInstance (3, textAlpha);
g2d.setComposite (acomp);

g2d.drawString(text, x, y);
}

g2d.dispose();
return texture;
}

private Point2D getEndPoint(int size, StripeDirection dir) {
  switch (dir) {
    case HORIZONTAL: return new Point2D.Float(size, 0);
    case VERTICAL: return new Point2D.Float(0, size);
    case DIAGONAL: return new Point2D.Float(size, size);
    default: return new Point2D.Float(size, 0);
  }
}

private Point2D getEndPoint(int width, int height, StripeDirection dir) {
  switch (dir) {

```

```

        case HORIZONTAL: return new Point2D.Float(width, 0);
        case VERTICAL: return new Point2D.Float(0, height);
        case DIAGONAL: return new Point2D.Float(width, height);
        default: return new Point2D.Float(width, 0);
    }
}

private Vector3 getLightDirection(Light light, Point3 worldPoint) {
    if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight)light).getDirection().normalize();
    } else if (light instanceof MuratPointLight) {
        return
        ((MuratPointLight)light).getPosition().subtract(worldPoint).normalize();
    } else if (light instanceof PulsatingPointLight) {
        return
        ((PulsatingPointLight)light).getPosition().subtract(worldPoint).normalize()
    ;
    } else if (light instanceof BioluminescentLight) {
        return
        ((BioluminescentLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof BlackHoleLight) {
        return
        ((BlackHoleLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight)light).getDirectionAt(worldPoint).normalize();
    } else {
        System.err.println("Warning: Unsupported light type: " +
light.getClass().getName());
        return null;
    }
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        return bgStartColor;
    }
}

```

```

Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);

Vector3 localNormal =
objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

Color textureColor = getTextureColor(localPoint, localNormal);
if (textureColor.getAlpha() == 0) {
    return new Color(0, 0, 0, 0);
}

double texR = textureColor.getRed() / 255.0;
double texG = textureColor.getGreen() / 255.0;
double texB = textureColor.getBlue() / 255.0;

double rCombined = 0.0;
double gCombined = 0.0;
double bCombined = 0.0;

if (light instanceof ElenaMuratAmbientLight) {
    double ambientIntensity = light.getIntensity();
    rCombined = texR * ambientIntensity * (light.getColor().getRed() /
255.0);
    gCombined = texG * ambientIntensity * (light.getColor().getGreen() /
255.0);
    bCombined = texB * ambientIntensity * (light.getColor().getBlue() /
255.0);
} else {
    Vector3 lightDir = getLightDirection(light, worldPoint);
    if (lightDir != null) {
        double diffuseFactor = Math.max(0, worldNormal.dot(lightDir));
        rCombined = texR * diffuseFactor * (light.getColor().getRed() /
255.0) * light.getIntensity();
        gCombined = texG * diffuseFactor * (light.getColor().getGreen() /
255.0) * light.getIntensity();
        bCombined = texB * diffuseFactor * (light.getColor().getBlue() /
255.0) * light.getIntensity();
    }
}

```

```

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
double specFactor = Math.pow(Math.max(0,
viewDir.dot(reflectDir)), 32);

rCombined += specFactor * (light.getColor().getRed() / 255.0);
gCombined += specFactor * (light.getColor().getGreen() / 255.0);
bCombined += specFactor * (light.getColor().getBlue() / 255.0);
}

}

}

return new Color(
(float)Math.min(1.0, Math.max(0.0, rCombined)),
(float)Math.min(1.0, Math.max(0.0, gCombined)),
(float)Math.min(1.0, Math.max(0.0, bCombined)),
textureColor.getAlpha() / 255.0f
);
}

/***
 * Maps a 3D point on the sphere to a 2D texture coordinate using
spherical mapping.
 * The texture is sampled with proper orientation, ensuring text appears
upright
 * when viewed from the front of the sphere.
 *
 * @param localPoint The point on the surface in object space.
 * @param localNormal The surface normal (unused here, kept for
interface).
 * @return The color sampled from the texture.
 */
private Color getTextureColor(Point3 localPoint, Vector3 localNormal) {
if (texture == null) return bgStartColor;

// Normalize direction vector from center to point
Vector3 dir = new Vector3(localPoint.x, localPoint.y,
localPoint.z).normalize();

```

```

// Convert to spherical coordinates
double phi = Math.atan2(dir.z, dir.x);           // -π to π
double theta = Math.asin(dir.y);                 // -π/2 to π/2

// Map to UV [0,1]
// U: Reverse the horizontal wrap so text appears correct
double u = 1.0 - (phi + Math.PI) / (2 * Math.PI); // Flip U horizontally
double v = (theta + Math.PI / 2) / Math.PI;        // V: top to bottom

// Flip V because BufferedImage has Y-down
v = 1.0 - v;

// Wrap U for seamless tiling
int texX = (int)(u * texture.getWidth());
texX = texX % texture.getWidth();
if (texX < 0) texX += texture.getWidth();

// Clamp V
int texY = (int)(v * texture.getHeight());
if (texY < 0 || texY >= texture.getHeight()) {
    return new Color(0, 0, 0, 0);
}

return new Color(texture.getRGB(texX, texY), true);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

```

```

    }

}

/***
EMShape sphere = new Sphere(1.2);

sphere.setTransform(Matrix4.translate(0, 1.2, 0));

Material material = new GradientTextMaterial(
Color.GREEN, Color.WHITE.darker (), //BG Colors
Color.RED, Color.BLUE,          // Gradient colors
"Takk",                      // Norwegian text
new Font("Arial", Font.BOLD, 200),// Font
GradientTextMaterial.StripeDirection.DIAGONAL, // Gradient direction
0.2, 1.0, 0.0,                // reflectivity, IOR, transparency
sphere.getInverseTransform(),   // object transform
0, 0 //x and y offset
);

sphere.setMaterial(material);
*/

```

```

// =====
// File: /net/elenamurat/material/HolographicDiffractionMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class HolographicDiffractionMaterial implements Material {
    private Matrix4 objectInverseTransform;
    private final double reflectivity;

```

```

public HolographicDiffractionMaterial(Matrix4 objectInverseTransform)
{
    this(objectInverseTransform, 0.7);
}

public HolographicDiffractionMaterial(Matrix4 objectInverseTransform,
double reflectivity) {
    this.objectInverseTransform = objectInverseTransform;
    this.reflectivity = clamp(reflectivity, 0.0, 1.0);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
    // Light information (guaranteed in 0-255 range)
    Color lightColor = light.getColor();
    double intensity = clamp(light.getIntensityAt(worldPoint), 0.0, 1.0);
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

    // Hologram pattern (guaranteed in 0.0-1.0 range)
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    double r = 0.5 + 0.5 * Math.sin(10 * localPoint.x +
    clamp(lightColor.getRed()/255.0, 0.0, 1.0));
    double g = 0.5 + 0.5 * Math.sin(7 * localPoint.y +
    clamp(lightColor.getGreen()/255.0, 0.0, 1.0));
    double b = 0.5 + 0.5 * Math.cos(6 * localPoint.z +
    clamp(lightColor.getBlue()/255.0, 0.0, 1.0));

    // Light effect (guaranteed in 0.0-1.0 range)
    double NdotL = clamp(worldNormal.dot(lightDir), 0.1, 1.0);
    Color base = ColorUtil.createColor(r, g, b);
}

```

```

        return ColorUtil.blendColors(base, lightColor, NdotL * intensity * 0.5);
    }

// General clamp method
private static double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

private static int clamp(int value, int min, int max) {
    return Math.max(min, Math.min(max, value));
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return clamp(1.3, 1.0,
2.5); }
@Override public double getTransparency() { return clamp(0.8, 0.0, 1.0);
}

}

```

```

// =====
// File: /net/elenamurat/material/WaterfallMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

public class WaterfallMaterial implements Material {
    private final Color baseColor;
    private final double flowSpeed;
    private final long startTime;

    public WaterfallMaterial() {

```

```

        this(new Color(135, 206, 250), 0.1);
    }

    public WaterfallMaterial(Color baseColor, double flowSpeed) {
        this.baseColor = baseColor != null ? baseColor : new Color(135, 206,
250);
        this.flowSpeed = Math.max(0.01, Math.min(1.0, flowSpeed));
        this.startTime = System.currentTimeMillis();
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
        // 1. Time-based animation
        double time = (System.currentTimeMillis() - startTime) * 0.001 *
flowSpeed;

        // 2. Base wave pattern
        double verticalWave = bound(Math.sin(point.y * 12 + time * 1.5) *
0.25, -0.25, 0.25);

        // 3. Cross turbulence
        double turbulence = Math.sin(point.x * 7 + time * 1.8) *
Math.cos(point.z * 5 - time * 2.2) *
0.15;

        // 4. Dynamic alpha
        double baseAlpha = 0.25 + 0.5 * (1 - bound(point.y, 0.1, 0.9));
        double alpha = bound(baseAlpha + verticalWave + turbulence, 0.15,
0.85);

        // 5. Foam density
        double foamInput = Math.sin(point.z * 8 - time * 4) * 0.6 +
Math.sin(point.y * 20 + time * 2) * 0.4;
        double foamIntensity = bound(foamInput, 0, 1);

        // 6. Foam effect
        double foamThreshold = 0.45;
        if (alpha > foamThreshold) {

```

```

        double foamFactor = Math.pow((alpha - foamThreshold) / (1 -
foamThreshold), 1.5);
        int foamValue = (int)(220 * foamFactor);

        return new Color(
            clamp(baseColor.getRed() * 0.6 + foamValue),
            clamp(baseColor.getGreen() * 0.7 + foamValue),
            clamp(baseColor.getBlue() * 0.8 + foamValue),
            (int)(alpha * 255)
        );
    }

// 7. Non-foam area
return new Color(
    clamp(baseColor.getRed() * (0.7 + turbulence * 0.3)),
    clamp(baseColor.getGreen() * (0.8 + turbulence * 0.2)),
    clamp(baseColor.getBlue() * (0.9 + turbulence * 0.1)),
    (int)(alpha * 255)
);
}
}

@Override
public double getReflectivity() {
    return 0.2; // Fixed reflectivity value
}

@Override
public double getIndexOfRefraction() {
    return 1.33; // Water's refractive index
}

@Override
public double getTransparency() {
    return 0.7; // Semi-transparent
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

```

```
// Helper methods
private int clamp(double value) {
    return (int) Math.max(0, Math.min(255, value));
}

private double bound(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

}
```

```
// =====
// File: /net/elenamurat/material/QuantumFieldMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

public class QuantumFieldMaterial implements Material {
    private final Color primaryColor;
    private final Color secondaryColor;
    private final double energy;
    private Matrix4 objectInverseTransform;
    private double time;
    private final double reflectivity=0.3;

    public QuantumFieldMaterial(Color primary, Color secondary,
        double energy, Matrix4 invTransform) {
        this.primaryColor = primary;
        this.secondaryColor = secondary;
        this.energy = Math.max(0.1, Math.min(5.0, energy));
        this.objectInverseTransform = invTransform;
    }
```

```

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

public void update(double deltaTime) {
    time += deltaTime * energy * 0.1;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
    Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
    Vector3 normal =
objectInverseTransform.transformNormal(worldNormal).normalize();

    // Quantum noise patterns
    double nx = localPoint.x * 2 + time;
    double ny = localPoint.y * 3;
    double nz = localPoint.z + time;
    double pattern1 = (Math.sin(nx * 12.9898 + ny * 78.233 + nz *
144.7212) * 43758.5453) -
        Math.floor(Math.sin(nx * 12.9898 + ny * 78.233 + nz * 144.7212) *
43758.5453);

    double pattern2 = (Math.cos(nx * 9.1234 + ny * 45.678 + nz * 98.765) *
54321.987) -
        Math.floor(Math.cos(nx * 9.1234 + ny * 45.678 + nz * 98.765) *
54321.987);

    double pattern = Math.sin(pattern1 * Math.PI * 2) * Math.cos(pattern2 *
Math.PI * 3);

    // Energy pulse
    double pulse = 0.5 + 0.5 * Math.sin(time * 2 + localPoint.length() * 5);
}

```

```

// Base color
Color baseColor = new Color(
    (int)(primaryColor.getRed() * (1-pattern) + secondaryColor.getRed() *
pattern * pulse),
    (int)(primaryColor.getGreen() * (1-pattern) +
secondaryColor.getGreen() * pattern * pulse),
    (int)(primaryColor.getBlue() * (1-pattern) + secondaryColor.getBlue() *
pattern * pulse)
);

// Lighting
Vector3 lightDir = light.getPosition().subtract(worldPoint).normalize();
double NdotL = Math.max(0, normal.dot(lightDir));
Color directLight = new Color(
    (int)(baseColor.getRed() * light.getColor().getRed() * NdotL *
light.getIntensityAt(worldPoint) / 255.0),
    (int)(baseColor.getGreen() * light.getColor().getGreen() * NdotL *
light.getIntensityAt(worldPoint) / 255.0),
    (int)(baseColor.getBlue() * light.getColor().getBlue() * NdotL *
light.getIntensityAt(worldPoint) / 255.0)
);

// Energy glow
double glow = Math.pow(pulse * 0.7 + 0.3, 2) * (0.5 + 0.5 * pattern);
Color energyGlow = new Color(
    (int)(secondaryColor.getRed() * light.getColor().getRed() * glow * *
energy * 0.3 / 255.0),
    (int)(secondaryColor.getGreen() * light.getColor().getGreen() * glow * *
energy * 0.3 / 255.0),
    (int)(secondaryColor.getBlue() * light.getColor().getBlue() * glow * *
energy * 0.3 / 255.0)
);

// Combine
int r = Math.min(255, directLight.getRed() + energyGlow.getRed());
int g = Math.min(255, directLight.getGreen() +
energyGlow.getGreen());
int b = Math.min(255, directLight.getBlue() + energyGlow.getBlue());
return new Color(r, g, b);

```

```
}
```

```
@Override public double getReflectivity() { return reflectivity; }  
@Override public double getIndexOfRefraction() { return 1.1; }  
@Override public double getTransparency() { return 0.2; }
```

```
}
```

```
/***
```

```
// Camera setup (common for all materials)  
rayTracer.setCameraPosition(new Point3(0, 0, 5));  
rayTracer.setLookAt(new Point3(0, 0, 0));  
rayTracer.setUpVector(new Vector3(0, 1, 0));  
rayTracer.setFov(45.0);
```

```
// Ambient light (soft lighting for entire scene)  
scene.addLight(new ElenaMuratAmbientLight(  
new Color(200, 220, 255), // Bluish ambient  
0.3 // Intensity  
));
```

```
// 1. Create sphere  
Sphere quantumSphere = new Sphere(0.8);  
quantumSphere.setTransform(Matrix4.translate(new Vector3(0, 1.5, -4)));
```

```
// 2. Create material (purple-effect quantum field)  
QuantumFieldMaterial quantumMat = new QuantumFieldMaterial(  
new Color(70, 0, 120), // Dark purple  
new Color(0, 200, 255), // Cyan  
3.5, // Energy level  
quantumSphere.getInverseTransform()  
);
```

```
// 3. Assign material  
quantumSphere.setMaterial(quantumMat);
```

```
// 4. Add to scene  
scene.addShape(quantumSphere);
```

```
// 5. Update for animation
void renderLoop() {
    double deltaTime = 0.016;
    quantumMat.update(deltaTime);
    // ... rendering operations
}

// 6. Special lighting
scene.addLight(new PulsatingPointLight(
    new Point3(0, 3, 2),
    new Color(150, 0, 255), // Purple light
    2.0,
    0.5 // Pulse effect speed
));
/*
*/
```

```
// =====  
// File: /net/elenamurat/material/StainedGlassMaterial.java  
// =====
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.*;  
import net.elena.murat.util.ColorUtil;
```

```
public class StainedGlassMaterial implements Material {  
    private final Color tint;  
    private final double roughness;  
    private Matrix4 objectInverseTransform;  
    private final double reflectivity=0.05;
```

```
public StainedGlassMaterial(Color tint, double roughness, Matrix4  
invTransform) {  
    // Start with clamping  
    this.tint = new Color(
```

```

        clamp(tint.getRed(), 0, 255),
        clamp(tint.getGreen(), 0, 255),
        clamp(tint.getBlue(), 0, 255)
    );
    this.roughness = clamp(roughness, 0.01, 1.0);
    this.objectInverseTransform = invTransform;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
    // Refraction direction
    double eta = 1.0 / getIndexOfRefraction();
    Vector3 viewDir = viewPos.subtract(worldPoint).normalize();
    double cosTheta = clamp(-worldNormal.dot(viewDir), -1.0, 1.0);
    double k = 1.0 - eta * eta * (1.0 - cosTheta * cosTheta);

    Vector3 refractedDir = k < 0 ? viewDir.reflect(worldNormal) :
        viewDir.scale(eta).add(worldNormal.scale(eta * cosTheta -
Math.sqrt(k)));
}

// Background color (sky blue, clamped)
Color bgColor = new Color(100, 150, 255);

// Transmitted color (0-255 garantili)
Color transmitted = ColorUtil.multiplyColors(
    bgColor,
    tint,
    0.7 // attenuation factor
);

// Specular highlights (0-255 garantili)
Vector3 lightDir = light.getPosition().subtract(worldPoint).normalize();

```

```

Vector3 halfway = lightDir.add(viewDir).normalize();
double NdotH = clamp(worldNormal.dot(halfway), 0.0, 1.0);
double specularIntensity = clamp(Math.pow(NdotH, 1.0/roughness) *
0.5, 0.0, 1.0);

Color specularColor = ColorUtil.multiplyColors(
    light.getColor(),
    new Color(255, 255, 255), // white highlight
    specularIntensity * light.getIntensityAt(worldPoint)
);

// Combine with clamping (0-255 guaranteed)
return ColorUtil.add(transmitted, specularColor);
}

// Helper methods
private static int clamp(int value, int min, int max) {
    return Math.max(min, Math.min(max, value));
}

private static double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.52; }
@Override public double getTransparency() { return 0.9; }

}

/***
// Camera setup (common for all materials)
rayTracer.setCameraPosition(new Point3(0, 0, 5));
rayTracer.setLookAt(new Point3(0, 0, 0));
rayTracer.setUpVector(new Vector3(0, 1, 0));
rayTracer.setFov(45.0);

// Ambient light (soft lighting for entire scene)
scene.addLight(new ElenaMuratAmbientLight(

```

```

new Color(200, 220, 255), // Bluish ambient
0.3                      // Intensity
));

// 1. Create plane instead of sphere (for glass panel effect)
Plane glassPane = new Plane();
glassPane.setTransform(Matrix4.rotateX(Math.PI/2).translate(0, 0, -7));

// 2. Create material (green glass)
StainedGlassMaterial glassMat = new StainedGlassMaterial(
new Color(100, 255, 150, 150), // Green color (RGBA)
0.1,                         // Surface roughness
glassPane.getInverseTransform()
);

// 3. Assign material
glassPane.setMaterial(glassMat);

// 4. Add to scene
scene.addShape(glassPane);

// 5. Backlight for illumination
scene.addLight(new ElenaDirectionalLight(
new Vector3(0, 0, 1),    // Light coming from behind
new Color(200, 220, 255), // Bluish white
1.5
));
*/

```

```

package net.elena.murat.material;

import java.awt.*;
import java.awt.geom.Point2D;
import java.awt.image.BufferedImage;

```

```
import java.util.Random;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

/***
 * HybridTextMaterial — Combines dielectric material properties with
text/image rendering.
 * Supports reflection, refraction, Fresnel effect, and textured text on
curved surfaces.
 * Fully compatible with scene.txt loading and RayTracer integration.
*/
public class HybridTextMaterial implements Material {
    private final Color glassTint = new Color(0.95f, 0.97f, 1.0f, 1.0f);

    // --- TEXTURE PROPERTIES ---
    private final String word;
    private final Color textColor;
    private final Color gradientColor;
    private final String gradientType;
    private final Color bgColor;
    private final String fontFamily;
    private final int fontStyle;
    private final int fontSize;
    private final int uOffset;
    private final int vOffset;
    private final BufferedImage imageObject;
    private final int imageWidth;
    private final int imageHeight;
    private final int imageUOffset;
    private final int imageVOffset;
    private final BufferedImage texture;

    // --- DIELECTRIC PROPERTIES ---
    private Color diffuseColor;
    private double indexOfRefraction;
    private double transparency;
    private double reflectivity;
```

```

private Color filterColorInside;
private Color filterColorOutside;
private Matrix4 objectTransform;
private final Random random;
private double currentReflectivity;
private double currentTransparency;

// --- PHONG LIGHTING PROPERTIES ---
private Color specularColor;
private double shininess;
private double ambientCoefficient;
private double diffuseCoefficient;
private double specularCoefficient;

/**
 * Full constructor — supports all text, dielectric, and lighting properties.
 */
public HybridTextMaterial(String word, Color textColor, Color
gradientColor,
String gradientType, Color bgColor,
String fontFamily, int fontStyle, int fontSize,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset,
Color diffuseColor, double ior, double transparency, double reflectivity,
Color filterColorInside, Color filterColorOutside,
Color specularColor, double shininess,
double ambientCoefficient, double diffuseCoefficient, double
specularCoefficient) {

    // Text properties
    this.word = convertToNorwegianText(word).replaceAll("_", " ");
    this.textColor = textColor;
    this.gradientColor = gradientColor;
    this.gradientType = gradientType != null ? gradientType : "horizontal";
    this.bgColor = bgColor;
    this.fontFamily = fontFamily.replaceAll("_", " ");
    this.fontStyle = fontStyle;
    this.fontSize = fontSize;
}

```

```
this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

// Dielectric properties
this.diffuseColor = diffuseColor;
this.indexOfRefraction = ior;
this.transparency = transparency;
this.reflectivity = reflectivity;
this.filterColorInside = filterColorInside;
this.filterColorOutside = filterColorOutside;

// Phong lighting properties
this.specularColor = specularColor;
this.shininess = shininess;
this.ambientCoefficient = ambientCoefficient;
this.diffuseCoefficient = diffuseCoefficient;
this.specularCoefficient = specularCoefficient;

// Internal
this.random = new Random();
this.currentReflectivity = reflectivity;
this.currentTransparency = transparency;
this.objectTransform = new Matrix4().identity();

// Generate texture with improved visibility
this.texture = createTexture();
}

/**
 * Simplified constructor with defaults — ideal for scene.txt
 */
public HybridTextMaterial(String word, Color textColor, String
fontFamily, int fontStyle, int fontSize) {
    this(word, textColor, null, "horizontal", new Color(0x00000000),
```

```

fontFamily, fontStyle, fontSize, 0, 0,
null, 0, 0, 0, 0,
new Color(0.9f, 0.9f, 0.9f), 1.5, 0.8, 0.1,
new Color(1.0f, 1.0f, 1.0f), new Color(1.0f, 1.0f, 1.0f),
Color.WHITE, 32.0, 0.1, 0.7, 0.7);
}

/***
 * Creates the texture image with the word drawn centered, optionally
with a gradient and background image.
 * The texture size is fixed at 1024x1024 pixels.
*/
private BufferedImage createTexture() {
    final int size = 1024;
    BufferedImage texture = new BufferedImage(size, size,
BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = texture.createGraphics();

    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

    g2d.setBackground(new Color(0, 0, 0, 0));
    g2d.clearRect(0, 0, size, size);

    // Optional background image
    if (imageObject != null) {
        int imgX = ((size - imageWidth) / 2) + imageUOffset;
        int imgY = ((size - imageHeight) / 2) + imageVOffset;

        AlphaComposite alphaComposite =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.95f);
        g2d.setComposite(alphaComposite);
        g2d.drawImage(imageObject, imgX, imgY, imageWidth, imageHeight,
null);

        g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_O
VER)); // Reset to default
    }
}

```

```
}

// Font setup
Font font;
try {
    font = new Font(fontFamily, fontStyle, fontSize);
} catch (Exception e) {
    font = new Font("Arial", fontStyle, fontSize); // Fallback
}
g2d.setFont(font);

FontMetrics fm = g2d.getFontMetrics();
int textWidth = fm.stringWidth(word);
int textHeight = fm.getHeight();
int ascent = fm.getAscent();

int x = ((size - textWidth) / 2) + uOffset;
int y = ((size - textHeight) / 2) + (ascent * 2) + (textHeight / 3) +
vOffset;

AlphaComposite textComposite =
AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 0.98f);
g2d.setComposite(textComposite);

// Apply gradient or solid color
if (gradientColor != null) {
    GradientPaint gradient = createGradient(x, y - ascent, textWidth,
textHeight);
    g2d.setPaint(gradient);
} else {
    g2d.setColor(textColor);
}

g2d.drawString(word, x, y);
g2d.dispose();

return texture;
}
```

```

private GradientPaint createGradient(float x, float y, float width, float
height) {
    switch (gradientType.toLowerCase()) {
        case "vertical":
            return new GradientPaint(x, y, textColor, x, y + height / 2,
gradientColor, true);
        case "diagonal":
            return new GradientPaint(x, y, textColor, x + width / 3, y + height / 5,
gradientColor, true);
        case "horizontal":
        default:
            return new GradientPaint(x, y, textColor, x + width / 3, y,
gradientColor, true);
    }
}

```

```

public static String convertToNorwegianText(String input) {
    if (input == null || input.isEmpty()) return input;
    String result = input;
    result = result.replace("AE", "\u00C6");
    result = result.replace("O/", "\u00D8");
    result = result.replace("A0", "\u00C5");
    result = result.replace("ae", "\u00E6");
    result = result.replace("o/", "\u00F8");
    result = result.replace("a0", "\u00E5");
    return result;
}

```

// --- MATERIAL INTERFACE: CORE LIGHTING + TEXTURE +
DIELECTRIC ---

```

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    // 1. Get texture color at this point
    Point3 localPoint = objectTransform.inverse().transformPoint(point);
    Color textureColor = getTextureColor(localPoint, normal);

    // 2. Check if this point is textured (text or image area)
    boolean isTexturedArea = textureColor.getAlpha() > 50;

```

```

// 3. Fresnel effect for dynamic properties
Vector3 viewDir = viewerPoint.subtract(point).normalize();
double fresnel = Vector3.calculateFresnel(viewDir, normal, 1.0,
indexOfRefraction);

this.currentReflectivity = Math.min(0.95, reflectivity + (fresnel * 0.4));
this.currentTransparency = Math.max(0.05, transparency * (1.0 - fresnel
* 0.3));

// 4. Different treatment for textured vs non-textured areas
if(isTexturedArea) {
    // TEXTURED AREAS - Bright and vibrant with contrast boost
    Color brightColor =
ColorUtil.enhanceBrightnessAndContrast(textureColor, 1.3f, 1.2f);

    // Ambient component - normal
    Color ambient = ColorUtil.multiplyColor(brightColor,
ambientCoefficient * light.getIntensity());

    // Diffuse component - normal
    Vector3 lightDir = light.getDirectionTo(point).normalize();
    double NdotL = Math.max(0.0, normal.dot(lightDir));
    Color diffuse = ColorUtil.multiplyColor(brightColor,
diffuseCoefficient * NdotL * light.getIntensity());

    // Specular component - normal
    Vector3 reflectDir = lightDir.reflect(normal);
    double RdotV = Math.max(0.0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess);
    Color specular = ColorUtil.multiplyColor(specularColor,
specularCoefficient * specFactor * light.getIntensity());

    // Combine components - prioritize texture
    Color result = ColorUtil.add(ambient, ColorUtil.add(diffuse,
specular));
    return ColorUtil.clampColor(result);
}
else {

```

```

// GLASS AREAS - Normal lighting but ensure brightness
Color baseColor = diffuseColor;

Vector3 lightDir = light.getDirectionTo(point).normalize();
double diffuseFactor = Math.max(0.4, normal.dot(lightDir));

Color diffuse = ColorUtil.multiplyColor(baseColor, diffuseFactor *
light.getIntensity());

// Normal specular for glass
Vector3 reflectDir = lightDir.reflect(normal);
double specularFactor = Math.pow(Math.max(0,
viewDir.dot(reflectDir)), 40);
Color specular = ColorUtil.multiplyColor(specularColor,
specularFactor * 0.4 * light.getIntensity());

// Combine with light glass tint (not too strong)
Color result = ColorUtil.add(diffuse, specular);
Color glassTint = new Color(0.98f, 0.99f, 1.0f); // Very subtle tint
result = ColorUtil.multiplyColors(result, glassTint);

return ColorUtil.clampColor(result);
}

private Color getTextureColor(Point3 localPoint, Vector3 worldNormal)
{
    if(texture == null) return textColor;

    Vector3 dir = worldNormal.normalize();
    double phi = Math.atan2(dir.z, dir.x);
    double theta = Math.asin(dir.y);

    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI);
    double v = (theta + Math.PI / 2) / Math.PI;
    v = 1.0 - v;

    u = (u + 0.25) % 1.0; // Offset for alignment
}

```

```
int texX = (int) (u * texture.getWidth());
texX = texX % texture.getWidth();
if (texX < 0) texX += texture.getWidth();

int texY = (int) (v * texture.getHeight());
if (texY < 0 || texY >= texture.getHeight()) {
    return new Color(0, 0, 0, 0);
}

return new Color(texture.getRGB(texX, texY), true);
}

// --- MATERIAL INTERFACE METHODS ---

@Override
public void setObjectTransform(Matrix4 tm) {
    this.objectTransform = (tm != null) ? tm : new Matrix4().identity();
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

@Override
public double getTransparency() {
    return currentTransparency;
}

@Override
public double getReflectivity() {
    return currentReflectivity;
}

// --- GETTERS & SETTERS ---
public Color getDiffuseColor() { return diffuseColor; }
public void setDiffuseColor(Color color) { this.diffuseColor = color; }

public Color getSpecularColor() { return specularColor; }
public void setSpecularColor(Color color) { this.specularColor = color; }
```

```

public double getShininess() { return shininess; }
public void setShininess(double shininess) { this.shininess = shininess; }

public Color getFilterColorInside() { return filterColorInside; }
public void setFilterColorInside(Color color) { this.filterColorInside =
color; }

public Color getFilterColorOutside() { return filterColorOutside; }
public void setFilterColorOutside(Color color) { this.filterColorOutside =
color; }

public void setIndexOfRefraction(double ior) { this.indexOfRefraction =
ior; }
public void setTransparency(double transparency) { this.transparency =
transparency; }
public void setReflectivity(double reflectivity) { this.reflectivity =
reflectivity; }

public BufferedImage getTexture() { return texture; }

@Override
public String toString() {
    return String.format(
        "HybridTextMaterial[word=%s, ior=%.2f, trans=%.2f, refl=%.2f,
shininess=%.1f]",
        word, indexOfRefraction, transparency, reflectivity, shininess
    );
}

}

// =====
// File: /net/elenamurat/material/HexagonalHoneycombMaterial.java
// =====

package net.elena.murat.material;

```

```

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

public class HexagonalHoneycombMaterial implements Material {
    private final Color primaryColor;
    private final Color secondaryColor;
    private final Color borderColor;
    private final double cellSize;
    private final double borderWidth;
    private final double ambientStrength;
    private final double specularStrength;
    private final double shininess;
    private final double reflectivity = 0.01;

    public HexagonalHoneycombMaterial(Color primary, Color secondary,
double cellSize, double borderWidth) {
        this(primary, secondary, Color.BLACK, cellSize, borderWidth, 0.2, 0.5,
32.0);
    }

    public HexagonalHoneycombMaterial(Color primary, Color secondary,
Color borderColor,
        double cellSize, double borderWidth,
        double ambientStrength, double specularStrength,
        double shininess) {
        this.primaryColor = primary;
        this.secondaryColor = secondary;
        this.borderColor = borderColor;
        this.cellSize = Math.max(0.1, cellSize);
        this.borderWidth = Math.max(0, Math.min(0.5, borderWidth));
        this.ambientStrength = Math.max(0, Math.min(1, ambientStrength));
        this.specularStrength = Math.max(0, Math.min(1, specularStrength));
        this.shininess = Math.max(1, shininess);
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 normal, Light light,

```

```

Point3 viewPos) {
    // Hexagonal coordinate calculation
    double size = cellSize * 2.0;
    double x = (worldPoint.x + 1000) / size; // +1000 to avoid negative
values
    double z = (worldPoint.z + 1000) / size;

    // Hexagonal grid coordinates
    double q = (x * Math.sqrt(3.0)/3.0 - z / 3.0);
    double r = z * 2.0/3.0;

    // Find hexagon center
    int q1 = (int)Math.round(q);
    int r1 = (int)Math.round(r);
    int s1 = (int)Math.round(-q - r);

    // Check if inside hexagon
    double dq = Math.abs(q - q1);
    double dr = Math.abs(r - r1);
    double ds = Math.abs(-q - r - s1);

    // Determine cell color
    boolean isBorder = (dq > (0.5 - borderWidth)) ||
(dr > (0.5 - borderWidth)) ||
(ds > (0.5 - borderWidth));

    if(isBorder) {
        return borderColor;
    }

    // Select cell color for pattern
    boolean isPrimary = (q1 + r1 + s1) % 2 == 0;
    Color baseColor = isPrimary ? primaryColor : secondaryColor;

    // Lighting calculations
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();
    double NdotL = Math.max(0.1, normal.dot(lightDir));
    double intensity = light.getIntensityAt(worldPoint);
}

```

```

// Ambient + Diffuse
int red = (int)(baseColor.getRed() * (ambientStrength + NdotL *
intensity * light.getColor().getRed()/255.0));
int green = (int)(baseColor.getGreen() * (ambientStrength + NdotL *
intensity * light.getColor().getGreen()/255.0));
int blue = (int)(baseColor.getBlue() * (ambientStrength + NdotL *
intensity * light.getColor().getBlue()/255.0));

return new Color(
    Math.max(0, Math.min(255, red)),
    Math.max(0, Math.min(255, green)),
    Math.max(0, Math.min(255, blue)))
);
}

```

```

private double smoothstep(double edge0, double edge1, double x) {
    x = Math.max(0, Math.min(1, (x - edge0)/(edge1 - edge0)));
    return x * x * (3 - 2 * x);
}

```

```

private int clamp(int value) {
    return Math.max(0, Math.min(255, value));
}

```

```

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.0; }

```

```

@Override
public void setObjectTransform(Matrix4 tm) {
}

```

```
}
```

```
/***
```

```
Plane honeycombPlane = new Plane(new Point3(0,0,0), new
Vector3(0,1,0));
```

```
Material honeycombMat = new HexagonalHoneycombMaterial(
new Color(255, 215, 0), // Gold yellow
```

```
new Color(255, 255, 150), // Light yellow  
0.5, // Cell size (scale factor)  
0.05 // Border thickness  
);  
honeycombPlane.setMaterial(honeycombMat);
```

```
Material honeycomb = new HexagonalHoneycombMaterial(  
new Color(255, 239, 153), // Light yellow (honey color)  
new Color(255, 204, 51), // Gold yellow  
new Color(50, 50, 50), // Dark gray borders  
0.3, // Cell size  
0.05, // Border thickness  
0.2, // Ambient strength  
0.3, // Specular  
16.0 // Shininess  
);  
*/
```

```
// ======  
// File: /net/elenamurat/material/MosaicMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.Light;  
import net.elena.murat.util.ColorUtil;  
  
public class MosaicMaterial implements Material {  
    private Color baseColor;  
    private Color tileColor;  
    private double tileSize;  
    private double groutWidth;  
    private double randomness;  
    private double transparency;
```

```
public MosaicMaterial(Color baseColor, Color tileColor,
    double tileSize, double groutWidth, double randomness) {
    this.baseColor = baseColor;
    this.tileColor = tileColor;
    this.tileSize = tileSize;
    this.groutWidth = groutWidth;
    this.randomness = randomness;
    this.transparency = calculateTransparency(baseColor);
}
```

```
public MosaicMaterial(Color baseColor, Color tileColor) {
    this(baseColor, tileColor, 0.3, 0.05, 0.2);
}
```

```
private double calculateTransparency(Color color) {
    int alpha = color.getAlpha();
    return 1.0 - ((double)alpha / 255.0);
}
```

```
@Override
public double getTransparency() {
    return transparency;
}
```

```
@Override
public double getReflectivity() {
    return 0.1;
}
```

```
@Override
public double getIndexOfRefraction() {
    return 1.3;
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    // Not needed for this material
}
```

```
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    double pattern = calculateMosaicPattern(point);
    return ColorUtil.blendColors(baseColor, tileColor, pattern);
}

private double calculateMosaicPattern(Point3 point) {
    // Add some randomness to prevent perfect alignment
    double randomOffset = (point.hashCode() % 1000) / 1000.0 *
randomness;

    // Calculate tile coordinates with randomness
    double x = (point.x + randomOffset) / tileSize;
    double y = (point.y + randomOffset) / tileSize;
    double z = (point.z + randomOffset) / tileSize;

    // Get fractional parts to determine tile position
    double fracX = x - Math.floor(x);
    double fracY = y - Math.floor(y);
    double fracZ = z - Math.floor(z);

    // Check if we're in the grout area
    if (fracX < groutWidth || fracX > (1.0 - groutWidth) ||
        fracY < groutWidth || fracY > (1.0 - groutWidth) ||
        fracZ < groutWidth || fracZ > (1.0 - groutWidth)) {
        return 0.0; // Base color (grout)
    }

    // Add some variation to tile color based on position
    double variation = ((int)(Math.floor(x) + Math.floor(y) + Math.floor(z))
% 3) * 0.1;

    return 0.9 + variation; // Tile color with slight variation
}

public Color getBaseColor() {
    return baseColor;
}
```

```
public void setBaseColor(Color baseColor) {  
    this.baseColor = baseColor;  
    this.transparency = calculateTransparency(baseColor);  
}  
  
public Color getTileColor() {  
    return tileColor;  
}  
  
public void setTileColor(Color tileColor) {  
    this.tileColor = tileColor;  
}  
  
public double getTileSize() {  
    return tileSize;  
}  
  
public void setTileSize(double tileSize) {  
    this.tileSize = tileSize;  
}  
  
public double getGroutWidth() {  
    return groutWidth;  
}  
  
public void setGroutWidth(double groutWidth) {  
    this.groutWidth = groutWidth;  
}  
  
public double getRandomness() {  
    return randomness;  
}  
  
public void setRandomness(double randomness) {  
    this.randomness = randomness;  
}  
}
```

```

// =====
// File: /net/elenamurat/material/BlackHoleMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

public class BlackHoleMaterial implements Material {
    private final Point3 singularity;
    private Matrix4 objectInverseTransform;
    private final double reflectivity=0.95;

    public BlackHoleMaterial(Matrix4 objectInverseTransform) {
        this(new Point3(0,0,0), objectInverseTransform);
    }

    public BlackHoleMaterial(Point3 singularity, Matrix4
objectInverseTransform) {
        this.singularity = singularity;
        this.objectInverseTransform = objectInverseTransform;
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectInverseTransform = tm;
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
        // Light informations
        Color lightColor = light.getColor();

```

```
double intensity = light.getIntensityAt(worldPoint);

// Distorsiyon
Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
double dist = localPoint.distance(singularity);
if(dist < 0.5) return Color.BLACK;

// Redding according to light color
double warp = Math.min(1, 0.3/(dist*dist));
int r = (int)(lightColor.getRed() * warp);
int g = (int)(lightColor.getGreen() * warp * 0.3);
int b = (int)(lightColor.getBlue() * warp * 0.1);

return new Color(
    Math.min(255, r),
    Math.min(255, g),
    Math.min(255, b)
);
}
```

```
@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 2.5; }
@Override public double getTransparency() { return 0.1; }
```

```
}
```

```
// =====
// File: /net/elenamurat/material/FractalBarkMaterial.java
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```
import net.elena.murat.math.*;
import net.elena.murat.light.*;
```

```

public class FractalBarkMaterial implements Material {
    private Matrix4 objectInverseTransform;
    private double roughness=0.1;
    private final double reflectivity=clamp(0.1 * roughness, 0.0, 1.0);

    public FractalBarkMaterial(Matrix4 objectInverseTransform) {
        this(objectInverseTransform, 0.7);
    }

    public FractalBarkMaterial(Matrix4 objectInverseTransform, double
roughness) {
        this.objectInverseTransform = objectInverseTransform;
        this.roughness = clamp(roughness, 0.0, 1.0);
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectInverseTransform = tm;
    }

    // Fractal pattern between 0.0-1.0
    private double fractalNoise(Point3 p) {
        return clamp(Math.abs(Math.sin(30*p.x) * Math.cos(20*p.z) *
Math.sin(5*p.y)), 0.0, 1.0);
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
        // Light information (guaranteed between 0-255 range)
        Color lightColor = light.getColor();
        double intensity = clamp(light.getIntensityAt(worldPoint), 0.0, 1.0);
        Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

        // Fractal pattern (guaranteed between 0.0-1.0 range)
        Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
        double n = fractalNoise(localPoint);
    }
}

```

```

// Base bark color (guaranteed between 0-255 range)
int r = clamp((int)(100 + 100*n * (lightColor.getRed()/255.0)), 0, 255);
int g = clamp((int)(70 + 50*n * (lightColor.getGreen()/255.0)), 0, 255);
int b = clamp((int)(40 + 20*n * (lightColor.getBlue()/255.0)), 0, 255);
Color base = new Color(r, g, b);

// Diffuse lighting (guaranteed between 0-255 range)
double NdotL = clamp(worldNormal.dot(lightDir), 0.1, 1.0);
return new Color(
    clamp((int)(base.getRed() * NdotL * intensity), 0, 255),
    clamp((int)(base.getGreen() * NdotL * intensity), 0, 255),
    clamp((int)(base.getBlue() * NdotL * intensity), 0, 255)
);
}

// Helper clamp methods
private static double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

private static int clamp(int value, int min, int max) {
    return Math.max(min, Math.min(max, value));
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return clamp(1.2, 1.0, 2.5);
}

@Override
public double getTransparency() {
    return 0.0;
}

```

```
}
```

```
// =====  
// File: /net/elenamurat/material/SultanKingMaterial.java  
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```
import net.elena.murat.math.*;
```

```
import net.elena.murat.light.*;
```

```
import net.elena.murat.util.ColorUtil;
```

```
public class SultanKingMaterial implements Material {
```

```
    private final Color goldColor;
```

```
    private final Color rubyColor;
```

```
    private final Color sapphireColor;
```

```
    private final double royaltyIntensity;
```

```
    private Matrix4 objectTransform;
```

```
    private final double ambientCoeff = 0.45;
```

```
    private final double diffuseCoeff = 0.7;
```

```
    private final double specularCoeff = 0.4;
```

```
    private final double shininess = 75.0;
```

```
    private final double reflectivity = 0.35;
```

```
    private final double ior = 1.8;
```

```
    private final double transparency = 0.05;
```

```
    public SultanKingMaterial() {
```

```
        this(new Color(0xFF, 0xD7, 0x00), new Color(0xDC, 0x14, 0x3C), new
```

```
        Color(0x00, 0x64, 0xCD), 0.75);
```

```
    }
```

```
    public SultanKingMaterial(Color goldColor, Color rubyColor, Color  
        sapphireColor, double royaltyIntensity) {
```

```
        this.goldColor = goldColor;
```

```
    this.rubyColor = rubyColor;
    this.sapphireColor = sapphireColor;
    this.royaltyIntensity = Math.max(0, Math.min(1, royaltyIntensity));
    this.objectTransform = Matrix4.identity();
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}
```

```
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateRoyalRegalia(objectPoint, worldNormal,
viewerPos);
```

```
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;
```

```
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);
```

```
    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }
```

```
    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);
```

```
    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
```

```

        double specFactor = Math.pow(RdotV, shininess) * props.intensity;
        Color specular = ColorUtil.multiplyColors(goldColor, props.color,
specularCoeff * specFactor);

        return ColorUtil.combineColors(ambient, diffuse, specular);
    }

private Color calculateRoyalRegalia(Point3 point, Vector3 normal,
Point3 viewerPos) {
    double x = point.x * 12.0;
    double y = point.y * 12.0;
    double z = point.z * 12.0;

    // Ottoman Sultan patterns (crescent, tughra, geometric)
    double ottoman1 = Math.sin(x * 2.0 + Math.cos(y * 1.5) * 3.0);
    double ottoman2 = Math.abs(Math.cos(x * 2.5 + y * 2.0) + Math.sin(y *
2.2 + z * 1.8));
    double ottoman3 = (Math.floor(x * 0.8) + Math.floor(y * 0.8)) % 2.2;

    // Viking King patterns (crown, runes, animal motifs)
    double viking1 = Math.sin(x * 1.8 + Math.sin(y * 1.2) * 2.5);
    double viking2 = Math.abs(Math.cos(x * 3.0 + y * 1.7) + Math.sin(y *
2.5 + z * 1.3));
    double viking3 = Math.sin(x * 2.2 + y * 1.9) * Math.cos(y * 1.4 + z *
1.1);

    // Royal insignia and emblem patterns
    double emblem1 = Math.sin(x * 3.5 + y * 2.8) + Math.cos(y * 2.5 + z *
2.0);
    double emblem2 = Math.abs(Math.sin(x * 4.0 + y * 3.0) * Math.cos(y *
2.2 + z * 1.7));

    // Royal fusion pattern
    double ottomanWeight = 0.5 * royaltyIntensity;
    double vikingWeight = 0.5 * royaltyIntensity;
    double emblemWeight = 0.2 * royaltyIntensity;

    double combinedPattern = (ottoman1 * 0.2 + ottoman2 * 0.15 +
ottoman3 * 0.1) * ottomanWeight +

```

```

(viking1 * 0.2 + viking2 * 0.15 + viking3 * 0.1) * vikingWeight +
(emblem1 * 0.05 + emblem2 * 0.05) * emblemWeight;

double normalizedPattern = (combinedPattern + 1.0) * 0.5;

// View-dependent metallic effects
Vector3 viewDir = viewerPos.subtract(point).normalize();
double viewEffect = Math.pow(Math.abs(viewDir.dot(normal)), 0.8);
double metallicEffect = 0.3 + viewEffect * 0.7;

// Royal material selection
if (normalizedPattern < 0.3) {
    // Gold base with intricate engravings
    double detail = normalizedPattern / 0.3;
    return ColorUtil.blendColors(
        ColorUtil.darkenColor(goldColor, 0.2),
        ColorUtil.lightenColor(goldColor, 0.1),
        detail * metallicEffect
    );
} else if (normalizedPattern < 0.5) {
    // Ruby inlays (Ottoman influence)
    double intensity = (normalizedPattern - 0.3) / 0.2;
    Color richRuby = ColorUtil.blendColors(
        rubyColor,
        ColorUtil.lightenColor(rubyColor, 0.3),
        intensity * viewEffect
    );
    return ColorUtil.addSpecularHighlight(richRuby, metallicEffect * 0.5);
} else if (normalizedPattern < 0.7) {
    // Sapphire accents (Viking influence)
    double intensity = (normalizedPattern - 0.5) / 0.2;
    Color deepSapphire = ColorUtil.blendColors(
        sapphireColor,
        ColorUtil.lightenColor(sapphireColor, 0.25),
        intensity * viewEffect
    );
    return ColorUtil.addSpecularHighlight(deepSapphire, metallicEffect *
0.6);
} else if (normalizedPattern < 0.85) {

```

```
// Gold-sapphire fusion areas
double intensity = (normalizedPattern - 0.7) / 0.15;
return ColorUtil.blendColors(
    sapphireColor,
    goldColor,
    intensity * metallicEffect
);
} else {
// Gold-ruby royal emblems
double intensity = (normalizedPattern - 0.85) / 0.15;
Color royalEmblem = ColorUtil.blendColors(
    goldColor,
    rubyColor,
    intensity * 0.7
);
return ColorUtil.addSpecularHighlight(royalEmblem, metallicEffect *
0.8);
}
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
```

```
// File: /net/elenamurat/material/TextureMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

public class TextureMaterial implements Material {
    private final BufferedImage texture;
    private final int width;
    private final int height;

    public TextureMaterial(BufferedImage texture) {
        if(texture == null) {
            throw new IllegalArgumentException("Texture cannot be null");
        }
        this.texture = texture;
        this.width = texture.getWidth();
        this.height = texture.getHeight();
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPos) {
        // Calculate texture coordinates in [0,1] range
        double u = clamp(point.x - Math.floor(point.x)); // x mod 1
        double v = 1.0 - clamp(point.y - Math.floor(point.y)); // y mod 1
        (flipped)

        // Calculate pixel position
        int x = (int)(u * (width - 1));
        int y = (int)(v * (height - 1));

        // Return RGB value directly (alpha ignored)
        return new Color(texture.getRGB(x, y));
    }
}
```

```
}

private double clamp(double value) {
    return Math.max(0.0, Math.min(1.0, value));
}

@Override
public double getReflectivity() {
    return 0.0; // No reflection
}

@Override
public double getIndexOfRefraction() {
    return 1.0; // Air refractive index
}

@Override
public double getTransparency() {
    return 0.0; // Fully opaque
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

}
```

```
// =====
// File: /net/elenamurat/material/LinearGradientMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;
```

```
public class LinearGradientMaterial implements Material {  
    private final Color topColor;  
    private final Color bottomColor;  
  
    public LinearGradientMaterial(Color topColor, Color bottomColor) {  
        this.topColor = topColor;  
        this.bottomColor = bottomColor;  
    }  
  
    @Override  
    public Color getColorAt(Point3 point, Vector3 normal, Light light,  
    Point3 viewerPos) {  
        double t = (point.y + 1.0) / 2.0; // y E [-1,1] -> [0,1]  
  
        int r = (int)(topColor.getRed() * t + bottomColor.getRed() * (1 - t));  
        int g = (int)(topColor.getGreen() * t + bottomColor.getGreen() * (1 -  
t));  
        int b = (int)(topColor.getBlue() * t + bottomColor.getBlue() * (1 - t));  
  
        r = ColorUtil.clampColorValue (r);  
        g = ColorUtil.clampColorValue (g);  
        b = ColorUtil.clampColorValue (b);  
  
        return new Color(r, g, b);  
    }  
  
    @Override  
    public double getReflectivity() {  
        return 0.0;  
    }  
  
    @Override  
    public double getIndexOfRefraction() {  
        return 1.0;  
    }  
  
    @Override  
    public double getTransparency() {
```

```

        return 0.0; //opaque
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
    }

}

// =====
// File: /net/lena/murat/material/DiffuseMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;

/**
 * DiffuseMaterial: A simple material that only performs diffuse
 * (Lambertian) reflection.
 * Implements only the diffuse part of the Phong model. Suitable for ideal
 * matte surfaces.
 * Does not include reflection, refraction or transparency.
 */
public class DiffuseMaterial implements Material {
    private final Color color;
    private final double diffuseCoefficient;
    private final double reflectivity;
    private final double ior;
    private final double transparency;

    /**
     * Full constructor method.
     *
     * @param color Diffuse surface color.

```

```

* @param diffuseCoefficient Diffuse reflection coefficient (between 0.0
- 1.0).
* @param reflectivity Reflection amount (0.0 = matte, 1.0 = mirror-
like).
* @param ior Index of refraction (e.g.: air=1.0, glass=1.5, water=1.33).
* @param transparency Transparency level (0.0 = opaque, 1.0 = fully
transparent).
*/
public DiffuseMaterial(Color color, double diffuseCoefficient,
    double reflectivity, double ior, double transparency) {
    this.color = color;
    this.diffuseCoefficient = clamp01(diffuseCoefficient);
    this.reflectivity = clamp01(reflectivity);
    this.ior = Math.max(1.0, ior);
    this.transparency = clamp01(transparency);
}

/**
* Simple constructor: takes only color, other values are default.
* Default values: diffuseCoefficient = 0.8, reflectivity = 0.0, ior = 1.0,
transparency = 0.0
*
* @param color Material's diffuse color.
*/
public DiffuseMaterial(Color color) {
    this(color, 0.8, 0.0, 1.0, 0.0);
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // Ambient light is added separately by the ray tracer (only once in total)
    // So here we only calculate diffuse.

    // Ambient only returns for ElenaMuratAmbientLight, but usually
handled separately
    if (light instanceof ElenaMuratAmbientLight) {
        // If ray tracer already adds ambient, just return the color here
        return color;
    }
}

```

```

}

// Light direction
Vector3 lightDir = getLightDirection(light, point);
if (lightDir == null) return Color.BLACK;

// Light intensity (including attenuation)
double intensity = getLightIntensity(light, point);
if (intensity <= 0) return Color.BLACK;

// Diffuse: N · L
double NdotL = Math.max(0.0, normal.dot(lightDir));
double contribution = diffuseCoefficient * NdotL * intensity;

int r = (int) (color.getRed() * contribution);
int g = (int) (color.getGreen() * contribution);
int b = (int) (color.getBlue() * contribution);

r = Math.min(255, r);
g = Math.min(255, g);
b = Math.min(255, b);

return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

```

```

@Override
public void setObjectTransform(Matrix4 tm) {
}

// --- Helper Methods ---

private double getLightIntensity(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight) light).getIntensity();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof SpotLight) {
        return ((SpotLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getAttenuatedIntensity(point);
    }
    return 1.0;
}

private Vector3 getLightDirection(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight)
light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {

```

```
        return ((BioluminescentLight)
light).getDirectionAt(point).normalize();
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getDirectionAt(point).normalize();
    }
    return null;
}
```

```
private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}
```

```
// Getter method
public Color getColor() {
    return color;
}
```

```
public double getDiffuseCoefficient() {
    return diffuseCoefficient;
}
```

```
}
```

```
// =====
// File: /net/lena/murat/material/TransparentColorMaterial.java
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```
import net.elena.murat.math.*;
import net.elena.murat.light.Light;
```

```
public class TransparentColorMaterial implements Material {
    private final double transparency;
```

```
private final double reflectivity;
private final double indexOfRefraction;

private final Color surfaceColor;

public TransparentColorMaterial(double transparency,
    double reflectivity, double ior) {
    this.transparency = Math.max(0, Math.min(1, transparency));
    this.reflectivity = Math.max(0, Math.min(1, reflectivity));
    this.indexOfRefraction = Math.max(1.0, ior);

    int alpha = (int)(this.transparency * 255);
    this.surfaceColor = new Color(0, 0, 0, alpha); // RGB=0,
    alpha=transparency
}

// Basic constructor
public TransparentColorMaterial(double transparency) {
    this(transparency, 0.0, 1.5);
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    return surfaceColor;
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return reflectivity;
}
```

```
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

@Override
public String toString() {
    return String.format("TransparentColorMaterial[trans=%.2f, refl=%.2f,
ior=%.2f]",
        transparency, reflectivity, indexOfRefraction);
}

}
```

```
// =====
// File: /net/elenamurat/material/TurkishTileMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class TurkishTileMaterial implements Material {
    private final Color baseColor;
    private final Color patternColor;
    private final double tileSize;
    private Matrix4 objectTransform;

    // Phong parameters
    private final double ambientCoeff = 0.4;
    private final double diffuseCoeff = 0.6;
    private final double specularCoeff = 0.8;
```

```

private final double shininess = 80.0;
private final Color specularColor = Color.WHITE;
private final double reflectivity = 0.3;
private final double ior = 1.5;
private final double transparency = 0.0;

public TurkishTileMaterial() {
    this(new Color(0, 102, 204), new Color(255, 255, 255), 2.0);
}

public TurkishTileMaterial(Color baseColor, Color patternColor, double
tileSize) {
    this.baseColor = baseColor;
    this.patternColor = patternColor;
    this.tileSize = Math.max(0.5, tileSize);
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Color surfaceColor = calculatePatternColor(worldPoint, worldNormal);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }
}

```

```

        double NdotL = Math.max(0, worldNormal.dot(props.direction));
        Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

        Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
        Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
        double RdotV = Math.max(0, reflectDir.dot(viewDir));
        double specFactor = Math.pow(RdotV, shininess) * props.intensity;
        Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);

        return ColorUtil.combineColors(ambient, diffuse, specular);
    }

private Color calculatePatternColor(Point3 worldPoint, Vector3 normal)
{
    Point3 localPoint =
objectTransform.inverse().transformPoint(worldPoint);

    // Simple UV mapping based on dominant normal
    double u, v;
    double absX = Math.abs(normal.x);
    double absY = Math.abs(normal.y);
    double absZ = Math.abs(normal.z);

    if (absX > absY && absX > absZ) {
        u = localPoint.z * tileSize;
        v = localPoint.y * tileSize;
    } else if (absY > absX && absY > absZ) {
        u = localPoint.x * tileSize;
        v = localPoint.z * tileSize;
    } else {
        u = localPoint.x * tileSize;
        v = localPoint.y * tileSize;
    }

    // Clean tile repetition
    double tileU = u - Math.floor(u);

```

```

double tileV = v - Math.floor(v);

return createCleanTurkishPattern(tileU, tileV);
}

private Color createCleanTurkishPattern(double u, double v) {
    // 1. CRISP BORDER (10% border)
    double border = 0.1;
    if (u < border || u > 1.0 - border || v < border || v > 1.0 - border) {
        return patternColor;
    }

    // 2. CLEAN CENTRAL MEDALLION
    double centerX = 0.5;
    double centerY = 0.5;
    double dist = Math.sqrt(Math.pow(u - centerX, 2) + Math.pow(v - centerY, 2));

    // Central circle
    if (dist < 0.15) {
        // Sharp 8-petal flower
        double angle = Math.atan2(v - centerY, u - centerX);
        double petal = Math.abs(Math.sin(4 * angle)); // 8 petals (sin(4θ) gives
8 peaks)

        if (petal > 0.9 && dist > 0.05) {
            return patternColor;
        }
    }

    // Solid center
    if (dist < 0.05) {
        return patternColor;
    }

    return baseColor;
}

// 3. SHARP GEOMETRIC LINES (no anti-aliasing)
// Diagonals

```

```

if (Math.abs(u - v) < 0.02 || Math.abs(u + v - 1) < 0.02) {
    return patternColor;
}

// Vertical/horizontal lines
if (Math.abs(u - 0.5) < 0.01 || Math.abs(v - 0.5) < 0.01) {
    return patternColor;
}

// 4. CORNER ELEMENTS (sharp and clean)
double[] cornerDists = {
    Math.sqrt(u*u + v*v),
    Math.sqrt((1-u)*(1-u) + v*v),
    Math.sqrt(u*u + (1-v)*(1-v)),
    Math.sqrt((1-u)*(1-u) + (1-v)*(1-v))
};

for (double cornerDist : cornerDists) {
    if (cornerDist < 0.3) {
        double cornerAngle = Math.atan2(
            v - (cornerDist < 0.3 ? 0 : 1),
            u - (cornerDist < 0.3 ? 0 : 1)
        );
        double star = Math.abs(Math.sin(5 * cornerAngle));

        if (star > 0.95 && cornerDist > 0.15) {
            return patternColor;
        }
    }
}

// 5. BASE COLOR (no texture noise)
return baseColor;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```
@Override  
public double getIndexOfRefraction() {  
    return ior;  
}  
  
@Override  
public double getTransparency() {  
    return transparency;  
}  
}
```

```
// ======  
// File: /net/lena/murat/material/PhongMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
  
import net.elena.murat.light.*;  
import net.elena.murat.math.*;  
  
/**  
 * PhongMaterial implements the Phong reflection model, which includes  
 * ambient, diffuse, and specular components. It also defines properties  
 * for reflectivity, index of refraction, and transparency for advanced  
 * ray tracing effects.  
 * This material fully implements the extended Material interface.  
 */  
public class PhongMaterial implements Material {  
    private final Color diffuseColor;  
    private final Color specularColor;  
    private final double shininess;  
    private final double ambientCoefficient;  
    private final double diffuseCoefficient;  
    private final double specularCoefficient;
```

```

private final double reflectivity;
private final double ior; // Index of Refraction
private final double transparency;

/***
 * Full constructor for PhongMaterial.
 * @param diffuseColor The base color of the material (diffuse component).
 * @param specularColor The color of the specular highlight.
 * @param shininess The shininess exponent for specular highlights.
 * Higher values make highlights smaller and more intense.
 * @param ambientCoefficient The ambient light contribution coefficient (0.0 - 1.0).
 * @param diffuseCoefficient The diffuse light contribution coefficient (0.0 - 1.0).
 * @param specularCoefficient The specular light contribution coefficient (0.0 - 1.0).
 * @param reflectivity The reflectivity coefficient (0.0 - 1.0).
 * @param ior The Index of Refraction for transparent materials (igual or greater than 1.0).
 * @param transparency The transparency coefficient (0.0 - 1.0).
 */
public PhongMaterial(Color diffuseColor, Color specularColor, double shininess,
    double ambientCoefficient, double diffuseCoefficient, double specularCoefficient,
    double reflectivity, double ior, double transparency) {
    this.diffuseColor = diffuseColor;
    this.specularColor = specularColor;
    this.shininess = shininess;
    this.ambientCoefficient = ambientCoefficient;
    this.diffuseCoefficient = diffuseCoefficient;
    this.specularCoefficient = specularCoefficient;
    this.reflectivity = clamp01(reflectivity);
    this.ior = Math.max(1.0, ior);
    this.transparency = clamp01(transparency);
}

/***

```

```

* Simplified constructor with default parameters.
* Uses white specular color, shininess of 32.0, and default coefficients.
* Default reflectivity = 0.0, IOR = 1.0, transparency = 0.0.
* @param diffuseColor The base color of the material.
*/
public PhongMaterial(Color diffuseColor) {
    this(diffuseColor, Color.WHITE, 32.0,
        0.1, 0.7, 0.7,
        0.0, 1.0, 0.0);
}

// --- GETTERS (for internal use) ---
public Color getDiffuseColor() { return diffuseColor; }
public Color getSpecularColor() { return specularColor; }
public double getShininess() { return shininess; }
public double getAmbientCoefficient() { return ambientCoefficient; }
public double getDiffuseCoefficient() { return diffuseCoefficient; }
public double getSpecularCoefficient() { return specularCoefficient; }

// --- MATERIAL INTERFACE IMPLEMENTATION ---

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    Color lightColor = light.getColor();
    double attenuatedIntensity = 0.0;

    // Ambient component
    int rAmbient = (int) (diffuseColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
    int gAmbient = (int) (diffuseColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
    int bAmbient = (int) (diffuseColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);

    // If light is ambient, return only ambient contribution
    if (light instanceof ElenaMuratAmbientLight) {
        return new Color(
            Math.min(255, rAmbient),

```

```

        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

// Get light direction
Vector3 lightDir = getLightDirection(light, point);
if (lightDir == null) return Color.BLACK;

// Get attenuated intensity based on light type
if (light instanceof MuratPointLight) {
    attenuatedIntensity = ((MuratPointLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof ElenaDirectionalLight) {
    attenuatedIntensity = ((ElenaDirectionalLight) light).getIntensity();
} else if (light instanceof PulsatingPointLight) {
    attenuatedIntensity = ((PulsatingPointLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof SpotLight) {
    attenuatedIntensity = ((SpotLight) light).getAttenuatedIntensity(point);
} else if (light instanceof BioluminescentLight) {
    attenuatedIntensity = ((BioluminescentLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof BlackHoleLight) {
    attenuatedIntensity = ((BlackHoleLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof FractalLight) {
    attenuatedIntensity = ((FractalLight)
light).getAttenuatedIntensity(point);
} else {
    System.err.println("Warning: Unsupported light type in
PhongMaterial: " + light.getClass().getName());
    return Color.BLACK;
}

// Diffuse component
double NdotL = Math.max(0, normal.dot(lightDir));
int rDiffuse = (int) (diffuseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);

```

```

int gDiffuse = (int) (diffuseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int) (diffuseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

// Specular component
Vector3 viewDir = viewerPos.subtract(point).normalize();
Vector3 reflectDir = lightDir.negate().reflect(normal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

int rSpecular = (int) (specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
int gSpecular = (int) (specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
int bSpecular = (int) (specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

// Combine diffuse and specular (ambient added separately in
RayTracer)
int finalR = Math.min(255, rDiffuse + rSpecular);
int finalG = Math.min(255, gDiffuse + gSpecular);
int finalB = Math.min(255, bDiffuse + bSpecular);

return new Color(finalR, finalG, finalB);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {

```

```

        return transparency;
    }

@Override
public void setObjectTransform(Matrix4 tm) {
}

// --- HELPER METHODS ---

/***
 * Clamps a double value between 0.0 and 1.0.
 * @param val The value to clamp.
 * @return Clamped value in [0.0, 1.0].
 */
private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}

/***
 * Calculates the normalized direction from the light source to the given
point.
 * @param light The light source.
 * @param point The point in world space.
 * @return Normalized light direction vector, or null if unsupported.
 */
private Vector3 getLightDirection(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight)
light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight)

```

```

        light).getDirectionAt(point).normalize();
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getDirectionAt(point).normalize();
    } else {
        System.err.println("Warning: Unknown light type in PhongMaterial: "
+ light.getClass().getName());
        return new Vector3(0, 1, 0); // Fallback direction
    }
}

// =====
// File: /net/elenamurat/material/RosemalingMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class RosemalingMaterial implements Material {
    private final Color backgroundColor;
    private final Color flowerColor;
    private final Color accentColor;
    private final double patternDensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.5;
    private final double diffuseCoeff = 0.85;
    private final double specularCoeff = 0.15;
    private final double shininess = 20.0;
    private final double reflectivity = 0.08;
}
```

```

private final double ior = 1.5;
private final double transparency = 0.0;

public RosemalingMaterial() {
    this(new Color(0x2F, 0x4F, 0x4F), new Color(0xFF, 0x69, 0xB4), new
Color(0xFF, 0xD7, 0x00), 0.65);
}

public RosemalingMaterial(Color backgroundColor, Color flowerColor,
Color accentColor, double patternDensity) {
    this.backgroundColor = backgroundColor;
    this.flowerColor = flowerColor;
    this.accentColor = accentColor;
    this.patternDensity = Math.max(0, Math.min(1, patternDensity));
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateRosemalingPattern(objectPoint);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {

```

```

    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(Color.WHITE, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateRosemalingPattern(Point3 point) {
    double x = point.x * 15.0;
    double y = point.y * 15.0;
    double z = point.z * 15.0;

    // Rosemaling flower and scroll patterns
    double flower1 = Math.sin(x * 2.0) * Math.cos(y * 2.0 + Math.sin(z *
1.0));
    double flower2 = Math.abs(Math.sin(x * 3.0 + y * 2.0) + Math.cos(y *
2.5 + z * 1.5));
    double scroll = Math.sin(x * 1.5 + y * 1.2) * Math.cos(y * 0.8 + z *
0.7);
    double leaves = Math.abs(Math.cos(x * 2.2 + y * 1.8 + z * 1.0));

    double combinedPattern = (flower1 * 0.3 + flower2 * 0.25 + scroll * *
0.25 + leaves * 0.2);
    double normalizedPattern = (combinedPattern + 1.0) * 0.5;

    if (normalizedPattern < patternDensity * 0.4) {
        // Flower centers
        return accentColor;
    } else if (normalizedPattern < patternDensity * 0.7) {

```

```
// Flower petals
    double intensity = (normalizedPattern - patternDensity * 0.4) /
(patternDensity * 0.3);
    return ColorUtil.blendColors(flowerColor,
ColorUtil.lightenColor(flowerColor, 0.3), intensity);
} else if (normalizedPattern < patternDensity) {
// Scrollwork and leaves
    double intensity = (normalizedPattern - patternDensity * 0.7) /
(patternDensity * 0.3);
    return ColorUtil.blendColors(accentColor, backgroundColor, intensity
* 0.5);
} else {
// Background with subtle texture
    double intensity = (normalizedPattern - patternDensity) / (1.0 -
patternDensity);
    return ColorUtil.addColorVariation(backgroundColor, intensity);
}
}
```

```
@Override
public double getReflectivity() {
    return reflectivity;
}
```

```
@Override
public double getIndexOfRefraction() {
    return ior;
}
```

```
@Override
public double getTransparency() {
    return transparency;
}
```

```
}
```

```
// =====
// File: /net/elenamurat/material/TransparentEmojiMaterial.java
```

```
// ======

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.Light;
import net.elena.murat.math.Matrix4;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

/**

 * Material that displays a transparent emoji/image texture on a surface with
 * a checkerboard background pattern that covers the entire surface uniformly.
 * The emoji/image texture respects UV scaling and offset parameters with optional repeating.
 * Designed for EmojiBillboard (quad in XY plane, Z=0).
 * Uses planar UV mapping on X and Y axes with practical scaling and offset.
 * Supports alpha channel, extra transparency, and checkerboard background.
 */

public class TransparentEmojiMaterial implements Material {
    private final BufferedImage image;
    private double transparency;
    private final Color checkerColor1;
    private final Color checkerColor2;
    private final double checkerSize;
    private final double uOffset;
    private final double vOffset;
    private final double uScale;
    private final double vScale;
    private final boolean isRepeatTexture;

    private Matrix4 objectInverseTransform;
    private double objectWidth = 2.0;
```

```
private double objectHeight = 2.0;

private final boolean isMessy;

/***
 * Full constructor with all parameters including UV scaling and repeat
option
 * @param image The RGBA BufferedImage (with alpha channel)
 * @param transparency Extra transparency (0.0 = none, 1.0 = fully
transparent)
 * @param checkerColor1 First color of the checkerboard pattern
 * @param checkerColor2 Second color of the checkerboard pattern
 * @param checkerSize Size of each checkerboard tile in UV coordinates
 * @param uOffset Horizontal offset for texture coordinates (0.0-1.0
range)
 * @param vOffset Vertical offset for texture coordinates (0.0-1.0 range)
 * @param uScale Horizontal scaling factor (1.0 = original size, 0.5 =
half size, 2.0 = double size)
 * @param vScale Vertical scaling factor (1.0 = original size, 0.5 = half
size, 2.0 = double size)
 * @param isRepeatTexture Whether to repeat the texture outside [0,1]
UV range
 */
public TransparentEmojiMaterial(BufferedImage image,
Color checkerColor1, Color checkerColor2,
double checkerSize, double uOffset, double vOffset,
double uScale, double vScale,
boolean isRepeatTexture,
boolean isMessy) {
this.image = image;
this.checkerColor1 = checkerColor1;
this.checkerColor2 = checkerColor2;
this.checkerSize = Math.max(0.01, checkerSize);
this.uOffset = uOffset;
this.vOffset = vOffset;
this.uScale = Math.max(0.01, uScale);
this.vScale = Math.max(0.01, vScale);
this.isRepeatTexture = isRepeatTexture;
this.isMessy = isMessy;
```

```
    this.objectInverseTransform = new Matrix4();
}

/***
 * Constructor with default checkerboard colors and UV scaling
 */
public TransparentEmojiMaterial(BufferedImage image,
    double checkerSize, double uScale, double vScale) {
    this(image,
        new Color(200, 200, 200), new Color(150, 150, 150),
        checkerSize, 0.0, 0.0, uScale, vScale, false, false);
}

/***
 * Constructor with default checkerboard size and UV scaling
 */
public TransparentEmojiMaterial(BufferedImage image,
    double uScale, double vScale) {
    this(image, 0.1, uScale, vScale);
}

/***
 * Constructor with default parameters (no scaling, no offset)
 */
public TransparentEmojiMaterial(BufferedImage image) {
    this(image, 0.1, 1.0, 1.0);
}

/***
 * Empty constructor for later setup
 */
public TransparentEmojiMaterial() {
    this(null, 0.1, 1.0, 1.0);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
```

```

// Extract object dimensions from transform matrix for proper scaling
// Assuming the object is a quad with original size 2x2 units (from -1 to
+1)
// The scale factors are in the columns of the transformation matrix
this.objectWidth = 2.0 * Math.sqrt(
    tm.get(0, 0) * tm.get(0, 0) +
    tm.get(1, 0) * tm.get(1, 0) +
    tm.get(2, 0) * tm.get(2, 0)
);

this.objectHeight = 2.0 * Math.sqrt(
    tm.get(0, 1) * tm.get(0, 1) +
    tm.get(1, 1) * tm.get(1, 1) +
    tm.get(2, 1) * tm.get(2, 1)
);
}

```

```

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // Transform world point to local object space
    Point3 localPoint = objectInverseTransform.transformPoint(point);

    // Base UV coordinates for checkerboard background (range [0,1])
    double u_bg = (localPoint.x + 1.0) * 0.5;
    double v_bg = (1.0 - localPoint.y) * 0.5;

    // Get checkerboard background color at these UVs
    Color backgroundColor = getBackgroundCheckerboardColor(u_bg,
v_bg);

    if (isMessy) {
        if (backgroundColor.getAlpha () < 6) {
            setTransparency (1.0);
            return backgroundColor;
        }
    }
}

```

```

// If no image, return background color immediately
if (image == null) {
    setTransparency(0.0);
    return backgroundColor;
}

// Base UV for texture
double u_tex = u_bg;
double v_tex = v_bg;

// Apply scale and offset (do NOT normalize offset)
double scaledU = u_tex / uScale + uOffset;
double scaledV = v_tex / vScale + vOffset;

double finalU, finalV;

if (isRepeatTexture) {
    // Wrap UVs for tiling
    finalU = scaledU - Math.floor(scaledU);
    finalV = scaledV - Math.floor(scaledV);
} else {
    // No tiling: if UV outside [0,1], return background color immediately
    if (scaledU < 0.0 || scaledU > 1.0 || scaledV < 0.0 || scaledV > 1.0) {
        setTransparency(0.0);
        return backgroundColor;
    }
    finalU = scaledU;
    finalV = scaledV;
}

// Convert UV to pixel coordinates
int x = (int) (finalU * (image.getWidth() - 1));
int y = (int) (finalV * (image.getHeight() - 1));

// Clamp pixel indices to valid range
x = Math.max(0, Math.min(image.getWidth() - 1, x));
y = Math.max(0, Math.min(image.getHeight() - 1, y));

// Get pixel ARGB

```

```

int argb = image.getRGB(x, y);
int alpha = (argb >> 24) & 0xFF;

// Extract RGB
int r = (argb >> 16) & 0xFF;
int g = (argb >> 8) & 0xFF;
int b = argb & 0xFF;

if (alpha == 0) {
    // Fully transparent pixel: show checkerboard background
    setTransparency(1.0);
    return backgroundColor;
} else {
    // Opaque or semi-transparent pixel: blend PNG color with background
    based on alpha

    setTransparency(0.0);

    float alphaF = alpha / 255f;

    // Simple alpha blending: result = alpha * PNG + (1 - alpha) *
    background
    float bgR = backgroundColor.getRed() / 255f;
    float bgG = backgroundColor.getGreen() / 255f;
    float bgB = backgroundColor.getBlue() / 255f;

    float outR = alphaF * (r / 255f) + (1 - alphaF) * bgR;
    float outG = alphaF * (g / 255f) + (1 - alphaF) * bgG;
    float outB = alphaF * (b / 255f) + (1 - alphaF) * bgB;

    return new Color(outR, outG, outB, 1.0f);
}

/***
 * Returns the checkerboard color at given UV coordinates.
 * Checkerboard pattern covers entire surface uniformly, ignoring scale
and offset.
 * @param u Horizontal UV coordinate [0,1]

```

```

* @param v Vertical UV coordinate [0,1]
* @return Checkerboard color at UV
*/
private Color getBackgroundCheckerboardColor(double u, double v) {
    int ix = (int) Math.floor(u / checkerSize);
    int iy = (int) Math.floor(v / checkerSize);
    return ((ix + iy) % 2 == 0) ? checkerColor1 : checkerColor2;
}

/**
 * Gets pixel color with bounds checking and alpha support
 * @param x X coordinate in image space
 * @param y Y coordinate in image space
 * @return Color with alpha channel, or transparent if out of bounds
*/
private Color getPixelColor(int x, int y) {
    if (image == null) return new Color(0, 0, 0, 0);
    if (x < 0 || x >= image.getWidth() || y < 0 || y >= image.getHeight()) {
        return new Color(0, 0, 0, 0);
    }
    int rgb = image.getRGB(x, y);
    return new Color(
        (rgb >> 16) & 0xFF,
        (rgb >> 8) & 0xFF,
        rgb & 0xFF,
        (rgb >> 24) & 0xFF
    );
}

@Override
public double getReflectivity() {
    return 0.0; // Non-reflective material
}

@Override
public double getIndexOfRefraction() {
    return 1.0; // No refraction (same as air)
}

```

```
@Override
public double getTransparency() {
    return transparency;
}

private void setTransparency(double tnw) {
    this.transparency = tnw;
}

/***
 * Gets the first checkerboard color
 * @return First checkerboard color
 */
public Color getCheckerColor1() {
    return checkerColor1;
}

/***
 * Gets the second checkerboard color
 * @return Second checkerboard color
 */
public Color getCheckerColor2() {
    return checkerColor2;
}

/***
 * Gets the checkerboard tile size
 * @return Checkerboard tile size in UV coordinates
 */
public double getCheckerSize() {
    return checkerSize;
}

/***
 * Gets the horizontal texture offset
 * @return U offset value
 */
public double getUOffset() {
    return uOffset;
}
```

```
}

/***
 * Gets the vertical texture offset
 * @return V offset value
 */
public double getVOffset() {
    return vOffset;
}

/***
 * Gets the horizontal texture scale factor
 * @return U scale factor
 */
public double getUScale() {
    return uScale;
}

/***
 * Gets the vertical texture scale factor
 * @return V scale factor
 */
public double getVScale() {
    return vScale;
}

/***
 * Checks if texture repeating is enabled
 * @return true if texture repeating is enabled, false otherwise
 */
public boolean isItRepeatTexture() {
    return true;
}

/***
 * Linear interpolation helper method
 * @param a Start value
 * @param b End value
 * @param t Interpolation factor [0,1]

```

```

* @return Interpolated value between a and b
*/
private double lerp(double a, double b, double t) {
    return a + t * (b - a);
}

}

// =====
// File: /net/elenamurat/material/NordicWoodMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class NordicWoodMaterial implements Material {
    private final Color woodColor;
    private final Color grainColor;
    private final double grainIntensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.45;
    private final double diffuseCoeff = 0.85;
    private final double specularCoeff = 0.18;
    private final double shininess = 25.0;
    private final double reflectivity = 0.09;
    private final double ior = 1.6;
    private final double transparency = 0.0;

    public NordicWoodMaterial() {
        this(new Color(0x8B, 0x5A, 0x2B), new Color(0x5D, 0x40, 0x35),
0.5);
    }
}

```

```
public NordicWoodMaterial(Color woodColor, Color grainColor, double  
grainIntensity) {  
    this.woodColor = woodColor;  
    this.grainColor = grainColor;  
    this.grainIntensity = Math.max(0, Math.min(1, grainIntensity));  
    this.objectTransform = Matrix4.identity();  
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4();  
    this.objectTransform = tm;  
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light  
light, Point3 viewerPos) {
```

```
    Point3 objectPoint =  
    objectTransform.inverse().transformPoint(worldPoint);
```

```
    Color surfaceColor = calculateWoodGrain(objectPoint);
```

```
    LightProperties props = LightProperties.getLightProperties(light,  
worldPoint);
```

```
    if (props == null) return surfaceColor;
```

```
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,  
ambientCoeff);
```

```
    if (light instanceof ElenaMuratAmbientLight) {  
        return ambient;  
    }
```

```
    double NdotL = Math.max(0, worldNormal.dot(props.direction));  
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,  
diffuseCoeff * NdotL * props.intensity);
```

```
    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
```

```

Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(new Color(0xFF, 0xEC,
0xCB), props.color, specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateWoodGrain(Point3 point) {
    double x = point.x * 12.0;
    double y = point.y * 12.0;
    double z = point.z * 12.0;

    // Wood grain pattern simulation
    double ringPattern = Math.sin(x * 0.8 + Math.sin(y * 0.3) * 2.0);
    double linePattern = Math.abs(Math.sin(y * 4.0 + Math.cos(x * 1.2) *
0.5));
    double noisePattern = Math.sin(x * 5.0 + y * 3.0 + z * 2.0) * 0.3;

    double combinedPattern = (ringPattern * 0.5 + linePattern * 0.3 +
noisePattern * 0.2);
    double normalizedPattern = (combinedPattern + 1.0) * 0.5;

    if (normalizedPattern < grainIntensity) {
        // Wood grain lines
        double intensity = normalizedPattern / grainIntensity;
        return ColorUtil.blendColors(woodColor, grainColor, intensity * 0.7);
    } else {
        // Base wood color with variation
        double intensity = (normalizedPattern - grainIntensity) / (1.0 -
grainIntensity);
        return addWoodVariation(woodColor, intensity);
    }
}

private Color addWoodVariation(Color baseColor, double variation) {
    double warmth = 0.9 + Math.sin(variation * 15.0) * 0.1;
    double darkness = 0.85 + Math.cos(variation * 8.0) * 0.15;
}

```

```
int r = (int)(baseColor.getRed() * warmth * darkness);
int g = (int)(baseColor.getGreen() * warmth * darkness * 0.95);
int b = (int)(baseColor.getBlue() * darkness * 0.9);

    return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}
```

```
// =====
// File: /net/elenamurat/material/FractalFireMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

public class FractalFireMaterial implements Material {
    private final int iterations;
```

```

private final double chaos;
private final long startTime;
private final double scale;
private final double speed;

public FractalFireMaterial(int iterations, double chaos, double scale,
double speed) {
    this.iterations = Math.max(5, Math.min(30, iterations)); // Increased
iterations for more detail
    this.chaos = Math.max(0.1, Math.min(2.0, chaos)); // Chaos
parameter has wider range
    this.scale = Math.max(0.5, Math.min(3.0, scale)); // Scale is better
adjusted
    this.speed = Math.max(0.1, Math.min(2.0, speed)); // Animation
speed added
    this.startTime = System.currentTimeMillis();
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Time-based animation (for smoother movement)
    double time = (System.currentTimeMillis() - startTime) * 0.0005 *
speed;

    // 2. Scale the point and offset from center (for more interesting
patterns)
    double x = (point.x - 0.5) * scale;
    double y = (point.y - 0.5) * scale;
    double z = (point.z - 0.5) * scale * 0.5; // Z-axis for 3D effect

    // 3. Dynamic chaos parameters (for more lively fire effect)
    double cx = -0.7 + Math.sin(time * 0.7) * chaos;
    double cy = 0.27 + Math.cos(time * 0.5) * chaos;
    double cz = Math.sin(time * 0.3) * chaos * 0.5;

    // 4. 3D Fractal calculation (Julia Set + Perlin noise-like variation)
    int i;
    for (i = 0; i < iterations; i++) {

```

```

double nx = x * x - y * y - z * z + cx;
double ny = 2 * x * y + cy;
double nz = 2 * x * z + cz;
x = nx;
y = ny;
z = nz;
if (x * x + y * y + z * z > 4) break;
}

// 5. Color palette (fire-like tones)
double ratio = (double) i / iterations;
int r = (int) (255 * Math.min(1, 0.3 + ratio * 3.0)); // Bright red/orange
int g = (int) (255 * Math.min(1, ratio * 1.5)); // Yellow tones
int b = (int) (255 * Math.min(1, ratio * 0.3)); // Dark red

// 6. Lighting (more realistic reflection)
if (light != null && light.getPosition() != null) {
    Vector3 lightDir = light.getPosition().subtract(point).normalize();
    double dot = Math.max(0.2, normal.dot(lightDir)); // Added minimum
lighting
    r = (int) (r * dot);
    g = (int) (g * dot);
    b = (int) (b * dot);
}

// 7. Final color adjustments (more vibrant colors)
r = Math.min(255, r + 20); // Slightly brighter
g = Math.min(255, g + 10);
b = Math.max(0, b - 10); // Reduce blue

return new Color(r, g, b);
}

@Override
public double getReflectivity() { return 0.1; } // Slight reflection
@Override
public double getIndexOfRefraction() { return 1.0; }
@Override
public double getTransparency() { return 0.0; }

```

```

@Override
public void setObjectTransform(Matrix4 tm) {
}

/**
// Example usage:
EMShape fireSphere = new Sphere()
.setMaterial(new FractalFireMaterial(
20,    // iterations (higher means more detailed)
1.2,   // chaos (higher means more "scattered" fire)
1.5,   // scale (smaller values give larger patterns)
0.8    // speed (1.0 = normal speed)
));

// Don't forget to add light!
scene.addLight(new PointLight(
new Point3(2, 5, 3), // Light position
new Color(255, 200, 150), // Warm color (white-yellow)
1.5 // Light intensity
));
*/

```

```

// =====
// File: /net/lena/murat/material/NordicWeaveMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class NordicWeaveMaterial implements Material {

```

```

private final Color primaryColor;
private final Color secondaryColor;
private final Color accentColor;
private final double patternScale;
private Matrix4 objectTransform;

private final double ambientCoeff = 0.5;
private final double diffuseCoeff = 0.9;
private final double specularCoeff = 0.1;
private final double shininess = 10.0;
private final double reflectivity = 0.05;
private final double ior = 1.5;
private final double transparency = 0.0;

public NordicWeaveMaterial() {
    this(new Color(0x8B, 0x45, 0x13), new Color(0x00, 0x64, 0x64), new
Color(0xDC, 0xDC, 0xDC), 4.0);
}

public NordicWeaveMaterial(Color primaryColor, Color secondaryColor,
Color accentColor, double patternScale) {
    this.primaryColor = primaryColor;
    this.secondaryColor = secondaryColor;
    this.accentColor = accentColor;
    this.patternScale = patternScale;
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

```

```

Color surfaceColor = calculateKilimPattern(objectPoint);

LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
if (props == null) return surfaceColor;

Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(accentColor, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateKilimPattern(Point3 point) {
    double x = point.x * patternScale;
    double y = point.y * patternScale;
    double z = point.z * patternScale;

    // Viking rune patterns combined with Turkish geometric patterns
    double pattern1 = Math.sin(x * 2.0) * Math.cos(y * 2.0);
    double pattern2 = Math.abs(Math.sin(x * 3.0 + y * 3.0));
    double pattern3 = Math.floor(x * 0.5) + Math.floor(y * 0.5);

    double combinedPattern = (pattern1 + pattern2 + pattern3 % 2.0) % 3.0;
}

```

```
if (combinedPattern < 1.0) {
    return primaryColor;
} else if (combinedPattern < 2.0) {
    return secondaryColor;
} else {
    return accentColor;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/elenamurat/material/SolidCheckerboardMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;
```

```
public final class SolidCheckerboardMaterial implements Material {  
    private final Color color1;  
    private final Color color2;  
    private final double size;  
    private Matrix4 objectInverseTransform;  
    private final double ambient;  
    private final double diffuse;  
  
    public SolidCheckerboardMaterial(Color color1, Color color2, double  
size,  
        double ambient, double diffuse,  
        Matrix4 objectInverseTransform) {  
        // Null-safe initialization  
        this.color1 = color1 != null ? color1 : new Color(100, 100, 100);  
        this.color2 = color2 != null ? color2 : new Color(200, 200, 200);  
        this.size = size > 0 ? size : 1.0;  
        this.objectInverseTransform = objectInverseTransform != null ?  
objectInverseTransform : new Matrix4();  
        this.ambient = Math.max(0, Math.min(1, ambient));  
        this.diffuse = Math.max(0, Math.min(1, diffuse));  
    }  
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4();  
    this.objectInverseTransform = tm;  
}
```

@Override

```
public Color getColorAt(Point3 point, Vector3 normal, Light light,  
Point3 viewPos) {
```

```
    // Null-safe point
```

```
    Point3 safePoint = point != null ? point : Point3.ORIGIN;
```

```
    Vector3 N = normal.normalize(); // normal zaten normalize edilmiş  
    olmalı ama emin olalım
```

```
// Spherical mapping: normal vektöründen u,v çıkar
```

```

double u = (Math.atan2(N.z, N.x) + Math.PI) / (2 * Math.PI); // 0 to 1
double v = (Math.asin(N.y) + Math.PI/2) / Math.PI;           // 0 to 1

// Tiles size
int tiles = (int) (10); // 10x10
int x = (int) Math.floor(u * tiles);
int y = (int) Math.floor(v * tiles);

Color baseColor = ((x + y) % 2 == 0) ? color1 : color2;

// Ambient if there is no light
if (light == null) {
    return ColorUtil.scale(baseColor, ambient);
}

// Diffuse lighting
Point3 lightPos = light.getPosition() != null ? light.getPosition() : new
Point3(0, 10, 0);
Vector3 lightDir = lightPos.subtract(safePoint).normalize();
double NdotL = Math.max(0, normal.dot(lightDir));

return ColorUtil.scale(baseColor, ambient + diffuse * NdotL *
light.getIntensity());
}

// Reflection closed
@Override public double getReflectivity() { return 0.0; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.0; }

}

// =====
// File: /net/elenamurat/material/AmberMaterial.java
// =====

package net.elena.murat.material;

```

```

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

public class AmberMaterial implements Material {
    private Color baseColor;
    private double transparency;
    private double reflectivity;
    private double indexOfRefraction;
    private Matrix4 objectTransform;
    private Matrix4 inverseObjectTransform;

    public AmberMaterial() {
        this.baseColor = new Color(255, 176, 56);
        this.transparency = 0.5;
        this.reflectivity = 0.25;
        this.indexOfRefraction = 1.52;
        this.objectTransform = Matrix4.identity();
        this.inverseObjectTransform = Matrix4.identity();
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPoint) {
        Point3 localPoint = inverseObjectTransform.transformPoint(point);
        Vector3 localNormal =
        inverseObjectTransform.inverseTransposeForNormal().transformVector(n
        ormal).normalize();

        Vector3 lightDir = light.getDirectionTo(point).normalize();
        double diffuse = Math.max(0, localNormal.dot(lightDir));

        Vector3 viewDir = viewerPoint.subtract(localPoint).normalize();
        double rim = Math.pow(1.0 - Math.max(0, localNormal.dot(viewDir)), 2.0);
    }
}

```

```
double ambient = 0.1;
double intensity = light.getIntensity();
double totalLight = ambient + intensity * (0.7 * diffuse + 0.3 * rim);

int r = (int) (baseColor.getRed() * totalLight);
int g = (int) (baseColor.getGreen() * totalLight);
int b = (int) (baseColor.getBlue() * totalLight);

return new Color(
    Math.min(255, Math.max(0, r)),
    Math.min(255, Math.max(0, g)),
    Math.min(255, Math.max(0, b)))
);
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) {
        this.objectTransform = Matrix4.identity();
    } else {
        this.objectTransform = tm;
    }
    this.inverseObjectTransform = this.objectTransform.inverse();
}
```

```
}
```

```
// =====  
// File: /net/elenamurat/material/MarbleMaterial.java  
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```
//custom
```

```
import net.elena.murat.math.*;
```

```
import net.elena.murat.light.*;
```

```
import net.elena.murat.util.ColorUtil;
```

```
/**
```

```
* A material that simulates marble with natural veining patterns.
```

```
* Uses Perlin noise for realistic marble texture generation.
```

```
*/
```

```
public class MarbleMaterial implements Material {
```

```
    private final Color baseColor; // Base color of the marble
```

```
    private final Color veinColor; // Color of the veins
```

```
    private final double scale; // Controls the size of the marble pattern
```

```
    private final double veinDensity; // Controls how prominent the veins  
    are (0.0 to 1.0)
```

```
    private final double turbulence; // Controls the complexity of the veins  
    private Matrix4 objectInverseTransform;
```

```
    // Lighting coefficients
```

```
    private final double ambientCoefficient;
```

```
    private final double diffuseCoefficient;
```

```
    private final double specularCoefficient;
```

```
    private final double shininess;
```

```
    private final double reflectivity;
```

```
    private final double ior;
```

```
    private final double transparency;
```

```

// Specular color for marble (cool white)
private final Color specularColor = new Color(240, 240, 255);

/***
 * Full constructor for MarbleMaterial.
 * @param baseColor Base color of the marble.
 * @param veinColor Color of the veins.
 * @param scale Controls the size of the marble pattern.
 * @param veinDensity Controls vein prominence (0.0 to 1.0).
 * @param turbulence Controls vein complexity (0.0 to 1.0).
 * @param ambientCoefficient Ambient light contribution.
 * @param diffuseCoefficient Diffuse light contribution.
 * @param specularCoefficient Specular light contribution.
 * @param shininess Shininess for specular highlights.
 * @param reflectivity Material's reflectivity (0.0 to 1.0).
 * @param ior Index of refraction.
 * @param transparency Material's transparency (0.0 to 1.0).
 * @param objectInverseTransform Inverse transform matrix of the
object.
*/
public MarbleMaterial(Color baseColor, Color veinColor, double scale,
double veinDensity, double turbulence,
double ambientCoefficient, double diffuseCoefficient, double
specularCoefficient,
double shininess, double reflectivity, double ior, double transparency,
Matrix4 objectInverseTransform) {
this.baseColor = baseColor;
this.veinColor = veinColor;
this.scale = scale;
this.veinDensity = Math.max(0, Math.min(1, veinDensity));
this.turbulence = Math.max(0, Math.min(1, turbulence));
this.objectInverseTransform = objectInverseTransform;

this.ambientCoefficient = ambientCoefficient;
this.diffuseCoefficient = diffuseCoefficient;
this.specularCoefficient = specularCoefficient;
this.shininess = shininess;
this.reflectivity = reflectivity;

```

```

this.ior = ior;
this.transparency = transparency;
}

/***
 * Simplified constructor with default coefficients.
 * @param baseColor Base color of the marble.
 * @param veinColor Color of the veins.
 * @param scale Controls the size of the marble pattern.
 * @param veinDensity Controls vein prominence.
 * @param turbulence Controls vein complexity.
 * @param objectInverseTransform Inverse transform matrix of the
object.
*/
public MarbleMaterial(Color baseColor, Color veinColor, double scale,
double veinDensity, double turbulence,
Matrix4 objectInverseTransform) {
this(baseColor, veinColor, scale, veinDensity, turbulence,
0.1, // ambientCoefficient
0.7, // diffuseCoefficient
0.2, // specularCoefficient
50.0, // shininess (marble is quite shiny)
0.15, // reflectivity (marble has some reflectivity)
1.5, // indexOfRefraction
0.05, // slight transparency
objectInverseTransform);
}

@Override
public void setObjectTransform(Matrix4 tm) {
if (tm == null) tm = new Matrix4 ();
this.objectInverseTransform = tm;
}

/***
 * Generates marble pattern using simulated Perlin noise.
 * @param localPoint Point in object's local space.
 * @return Marble pattern color.
*/

```

```

private Color getMarbleColor(Point3 localPoint) {
    // Scale coordinates
    double x = localPoint.x * scale;
    double y = localPoint.y * scale;
    double z = localPoint.z * scale;

    // Create turbulence pattern
    double noise = turbulence(x, y, z, turbulence);

    // Create sine wave pattern that will be distorted by noise
    double marblePattern = Math.sin(x + noise * 10.0) * 0.5 + 0.5;

    // Apply vein density to control how prominent veins are
    marblePattern = Math.pow(marblePattern, 1.0 + (veinDensity * 3.0));

    // Blend between base and vein color
    return ColorUtil.blendColors(baseColor, veinColor, marblePattern);
}

/**
 * Simple turbulence function to create natural-looking patterns.
 */
private double turbulence(double x, double y, double z, double
turbulenceFactor) {
    double t = 0.0;
    double size = 0.5;

    while (size >= 0.01) {
        t += Math.abs(improvedNoise(x/size, y/size, z/size)) * size;
        size /= 2.0;
    }

    return t * turbulenceFactor;
}

/**
 * Improved Perlin noise function for better pattern generation.
 */
private double improvedNoise(double x, double y, double z) {

```

```

// This is a simplified version of Perlin noise
// In a full implementation, you'd want a proper noise function
int xi = (int)Math.floor(x) & 255;
int yi = (int)Math.floor(y) & 255;
int zi = (int)Math.floor(z) & 255;

x -= Math.floor(x);
y -= Math.floor(y);
z -= Math.floor(z);

double u = fade(x);
double v = fade(y);
double w = fade(z);

// Hash coordinates
int a = xi+yi*256+zi*256*256;
int b = a+1;
int aa = a%256;
int ab = (a+1)%256;
int ba = (a+256)%256;
int bb = (a+257)%256;

// Blend everything
double lerp1 = lerp(u, grad(aa, x, y, z), grad(ab, x-1, y, z));
double lerp2 = lerp(u, grad(ba, x, y-1, z), grad(bb, x-1, y-1, z));
double lerp3 = lerp(v, lerp1, lerp2);

return lerp3;
}

private double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
private double lerp(double t, double a, double b) { return a + t * (b - a); }
private double grad(int hash, double x, double y, double z) {
    int h = hash & 15;
    double u = h<8 ? x : y;
    double v = h<4 ? y : h==12||h==14 ? x : z;
    return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
}

```

```

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        System.err.println("Error: MarbleMaterial's inverse transform is null.
Returning black.");
        return Color.BLACK;
    }

    // Transform to local space
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    if (localNormal == null) {
        System.err.println("Error: MarbleMaterial's normal transform matrix is
null or invalid. Returning black.");
        return Color.BLACK;
    }

    // Get base marble color
    Color marbleBaseColor = getMarbleColor(localPoint);

    // Lighting calculation (same structure as other materials)
    Color lightColor = light.getColor();
    double attenuatedIntensity = 0.0;

    // Ambient component
    int rAmbient = (int)(marbleBaseColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
    int gAmbient = (int)(marbleBaseColor.getGreen() * ambientCoefficient
* lightColor.getGreen() / 255.0);
    int bAmbient = (int)(marbleBaseColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);

    if (light instanceof ElenaMuratAmbientLight) {
        return new Color(

```

```

        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

Vector3 lightDirection;
if (light instanceof MuratPointLight) {
    MuratPointLight pLight = (MuratPointLight) light;
    lightDirection = pLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = pLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof ElenaDirectionalLight) {
    ElenaDirectionalLight dLight = (ElenaDirectionalLight) light;
    lightDirection = dLight.getDirection().negate().normalize();
    attenuatedIntensity = dLight.getIntensity();
} else if (light instanceof PulsatingPointLight) {
    PulsatingPointLight ppLight = (PulsatingPointLight) light;
    lightDirection =
        ppLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = ppLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof SpotLight) {
    SpotLight sLight = (SpotLight) light;
    lightDirection = sLight.getDirectionAt(worldPoint);
    attenuatedIntensity = sLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BioluminescentLight) {
    BioluminescentLight bLight = (BioluminescentLight) light;
    lightDirection = bLight.getDirectionAt(worldPoint);
    attenuatedIntensity = bLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BlackHoleLight) {
    BlackHoleLight bhLight = (BlackHoleLight) light;
    lightDirection = bhLight.getDirectionAt(worldPoint);
    attenuatedIntensity = bhLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof FractalLight) {
    FractalLight fLight = (FractalLight) light;
    lightDirection = fLight.getDirectionAt(worldPoint);
    attenuatedIntensity = fLight.getAttenuatedIntensity(worldPoint);
} else {
    System.err.println("Warning: Unknown or unsupported light type for
MarbleMaterial shading: " + light.getClass().getName());
}

```

```

    return Color.BLACK;
}

// Diffuse component
double NdotL = Math.max(0, worldNormal.dot(lightDirection));
int rDiffuse = (int)(marbleBaseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int)(marbleBaseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int)(marbleBaseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

// Specular component
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectionVector =
lightDirection.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectionVector.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

int rSpecular = (int)(specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
int gSpecular = (int)(specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
int bSpecular = (int)(specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

// Sum up all components
int finalR = Math.min(255, rAmbient + rDiffuse + rSpecular);
int finalG = Math.min(255, gAmbient + gDiffuse + gSpecular);
int finalB = Math.min(255, bAmbient + bDiffuse + bSpecular);

return new Color(finalR, finalG, finalB);
}

```

@Override
public double getReflectivity() { return reflectivity; }

@Override
public double getIndexOfRefraction() { return ior; }

```
@Override
public double getTransparency() { return transparency; }

}

// =====
// File: /net/elenamurat/material/PhongTextMaterial.java
// =====

package net.elena.murat.material;

import java.awt.*;
import java.awt.geom.Point2D;
import java.awt.image.BufferedImage;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

/**
 * PhongTextMaterial - Combines text/image rendering capability with
 * Phong lighting model.
 * Renders text or images on a surface with ambient, diffuse, and specular
 * lighting.
 */
public class PhongTextMaterial implements Material {

    // --- TEXTURE PROPERTIES (from TextDielectricMaterial) ---
    private final String word;
    private final Color textColor;
    private final Color gradientColor;
    private final String gradientType;
    private final Color bgColor;
    private final String fontFamily;
    private final int fontStyle;
    private final int fontSize;
    private final int uOffset;
```

```

private final int vOffset;
private final BufferedImage imageObject;
private final int imageWidth;
private final int imageHeight;
private final int imageUOffset;
private final int imageVOffset;
private final BufferedImage texture;

// --- PHONG MATERIAL PROPERTIES ---
private final Color diffuseColor; // Base color (can be overridden by
texture)
private final Color specularColor;
private final double shininess;
private final double ambientCoefficient;
private final double diffuseCoefficient;
private final double specularCoefficient;
private final double reflectivity;
private final double ior;
private final double transparency;
private Matrix4 objectTransform;

/**
 * Full constructor with all text and Phong properties
 */
public PhongTextMaterial(String word, Color textColor, Color
gradientColor,
String gradientType, Color bgColor,
String fontFamily, int fontStyle, int fontSize,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset,
Color diffuseColor, Color specularColor, double shininess,
double ambientCoefficient, double diffuseCoefficient, double
specularCoefficient,
double reflectivity, double ior, double transparency) {

    // Text properties
    this.word = convertToNorwegianText(word).replaceAll("_", " ");
    this.textColor = textColor;
}

```

```

this.gradientColor = gradientColor;
this.gradientType = gradientType != null ? gradientType : "horizontal";
this.bgColor = bgColor;
this.fontFamily = fontFamily.replaceAll("_", " ");
this.fontSize = fontSize;
this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

// Phong properties
this.diffuseColor = diffuseColor;
this.specularColor = specularColor;
this.shininess = shininess;
this.ambientCoefficient = ambientCoefficient;
this.diffuseCoefficient = diffuseCoefficient;
this.specularCoefficient = specularCoefficient;
this.reflectivity = clamp01(reflectivity);
this.ior = Math.max(1.0, ior);
this.transparency = clamp01(transparency);
this.objectTransform = Matrix4.identity();

// Generate texture
this.texture = createTexture();
}

/**
 * Simplified constructor with defaults
 */
public PhongTextMaterial(String word, Color textColor, String
fontFamily, int fontStyle, int fontSize) {
    this(word, textColor, null, "horizontal", new Color(0x00000000),
        fontFamily, fontStyle, fontSize, 0, 0,
        null, 0, 0, 0, 0,
        new Color(0.9f, 0.9f, 0.9f), Color.WHITE, 32.0,

```

```

    0.1, 0.7, 0.7,
    0.0, 1.0, 0.0);
}

/***
 * Creates the texture image with the word drawn centered, optionally
with a gradient and background image.
 * The texture size is fixed at 1024x1024 pixels.
*/
private BufferedImage createTexture() {
    final int size = 1024;
    BufferedImage texture = new BufferedImage(size, size,
BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = texture.createGraphics();

    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

    g2d.setBackground(new Color(0, 0, 0, 0));
    g2d.clearRect(0, 0, size, size);

    // Optional background image
    if (imageObject != null) {
        int imgX = ((size - imageWidth) / 2) + imageUOffset;
        int imgY = ((size - imageHeight) / 2) + imageVOffset;
        g2d.drawImage(imageObject, imgX, imgY, imageWidth, imageHeight,
null);
    }

    // Font setup
    Font font;
    try {
        font = new Font(fontFamily, fontStyle, fontSize);
    } catch (Exception e) {
        font = new Font("Arial", fontStyle, fontSize); // Fallback
    }
    g2d.setFont(font);
}

```

```

FontMetrics fm = g2d.getFontMetrics();
int textWidth = fm.stringWidth(word);
int textHeight = fm.getHeight();
int ascent = fm.getAscent();

int x = ((size - textWidth) / 2) + uOffset;
int y = ((size - textHeight) / 2) + (ascent * 2) + (textHeight / 3) +
vOffset;

// Apply gradient or solid color
if (gradientColor != null) {
    GradientPaint gradient = createGradient(x, y - ascent, textWidth,
textHeight);
    g2d.setPaint(gradient);
} else {
    g2d.setColor(textColor);
}

g2d.drawString(word, x, y);
g2d.dispose();

return texture;
}

private GradientPaint createGradient(float x, float y, float width, float
height) {
    switch (gradientType.toLowerCase()) {
        case "vertical":
            return new GradientPaint(x, y, textColor, x, y + height / 2,
gradientColor, true);
        case "diagonal":
            return new GradientPaint(x, y, textColor, x + width / 3, y + height / 5,
gradientColor, true);
        case "horizontal":
        default:
            return new GradientPaint(x, y, textColor, x + width / 3, y,
gradientColor, true);
    }
}

```

```
}
```

```
public static String convertToNorwegianText(String input) {
    if (input == null || input.isEmpty()) return input;
    String result = input;
    result = result.replace("AE", "\u00C6");
    result = result.replace("O/", "\u00D8");
    result = result.replace("A0", "\u00C5");
    result = result.replace("ae", "\u00E6");
    result = result.replace("o/", "\u00F8");
    result = result.replace("a0", "\u00E5");
    return result;
}

// --- PHONG LIGHTING WITH TEXTURE ---
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Get texture color at this point
    Point3 localPoint = objectTransform.inverse().transformPoint(point);
    Color textureColor = getTextureColor(localPoint, normal);

    // 2. If texture has alpha=0, use diffuseColor (base)
    if (textureColor.getAlpha() == 0) {
        textureColor = diffuseColor;
    }

    // 3. Apply Phong lighting using textureColor as diffuse base
    Color lightColor = light.getColor();
    double attenuatedIntensity = 0.0;

    // Ambient component
    int rAmbient = (int) (textureColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
    int gAmbient = (int) (textureColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
    int bAmbient = (int) (textureColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);
```

```

// If light is ambient, return only ambient contribution
if (light instanceof ElenaMuratAmbientLight) {
    return new Color(
        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

// Get light direction
Vector3 lightDir = getLightDirection(light, point);
if (lightDir == null) return Color.BLACK;

// Get attenuated intensity based on light type
if (light instanceof MuratPointLight) {
    attenuatedIntensity = ((MuratPointLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof ElenaDirectionalLight) {
    attenuatedIntensity = ((ElenaDirectionalLight) light).getIntensity();
} else if (light instanceof PulsatingPointLight) {
    attenuatedIntensity = ((PulsatingPointLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof SpotLight) {
    attenuatedIntensity = ((SpotLight) light).getAttenuatedIntensity(point);
} else if (light instanceof BioluminescentLight) {
    attenuatedIntensity = ((BioluminescentLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof BlackHoleLight) {
    attenuatedIntensity = ((BlackHoleLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof FractalLight) {
    attenuatedIntensity = ((FractalLight)
light).getAttenuatedIntensity(point);
} else {
    System.err.println("Warning: Unsupported light type in
PhongTextMaterial: " + light.getClass().getName());
    return Color.BLACK;
}

```

```

// Diffuse component
double NdotL = Math.max(0, normal.dot(lightDir));
int rDiffuse = (int) (textureColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int) (textureColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int) (textureColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

// Specular component
Vector3 viewDir = viewerPos.subtract(point).normalize();
Vector3 reflectDir = lightDir.negate().reflect(normal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

int rSpecular = (int) (specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
int gSpecular = (int) (specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
int bSpecular = (int) (specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

// Combine components
int finalR = Math.min(255, rAmbient + rDiffuse + rSpecular);
int finalG = Math.min(255, gAmbient + gDiffuse + gSpecular);
int finalB = Math.min(255, bAmbient + bDiffuse + bSpecular);

return new Color(finalR, finalG, finalB);
}

private Color getTextureColor(Point3 localPoint, Vector3 worldNormal)
{
    if(texture == null) return new Color(0, 0, 0, 0);

    Vector3 dir = worldNormal.normalize();
    double phi = Math.atan2(dir.z, dir.x);
    double theta = Math.asin(dir.y);

    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI);

```

```
double v = (theta + Math.PI / 2) / Math.PI;
v = 1.0 - v;

u = (u + 0.25) % 1.0; // Offset for better alignment

int texX = (int) (u * texture.getWidth());
texX = texX % texture.getWidth();
if (texX < 0) texX += texture.getWidth();

int texY = (int) (v * texture.getHeight());
if (texY < 0 || texY >= texture.getHeight()) {
    return new Color(0, 0, 0, 0);
}

return new Color(texture.getRGB(texX, texY), true);
}

// --- MATERIAL INTERFACE ---
@Override
public void setObjectTransform(Matrix4 tm) {
    this.objectTransform = (tm != null) ? tm : new Matrix4();
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

// --- GETTERS ---
```

```

public Color getDiffuseColor() { return diffuseColor; }
public Color getSpecularColor() { return specularColor; }
public double getShininess() { return shininess; }
public BufferedImage getTexture() { return texture; }

private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}

private Vector3 getLightDirection(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight)
light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight)
light).getDirectionAt(point).normalize();
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getDirectionAt(point).normalize();
    } else {
        System.err.println("Warning: Unknown light type in
PhongTextMaterial: " + light.getClass().getName());
        return new Vector3(0, 1, 0); // Fallback direction
    }
}

```

@Override

```

public String toString() {
    return String.format("PhongTextMaterial[text='%s', diffuse=%s,
shininess=%.1f]",
```

```

        word, diffuseColor, shininess);
    }

}

// =====
// File: /net/elenamurat/material/ImageTextureMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.Light;
import net.elena.murat.light.LightProperties;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;
//import net.elena.murat.util.ImageUtils3D;

/**
 * ImageTextureMaterial applies a loaded BufferedImage as a texture to a
 * surface.
 * It uses planar UV mapping based on the object's local space and applies
 * a Phong-like
 * lighting model (Ambient + Diffuse + Specular).
 * This material fully implements the extended Material interface with
 * proper texture wrapping
 * to eliminate black gaps between texture tiles.
 */
public class ImageTextureMaterial implements Material {
    private final BufferedImage image;
    private final double uScale;
    private final double vScale;
    private final double uOffset;
    private final double vOffset;

    // Phong material properties

```

```
private final Color specularColor;
private final double shininess;
private final double ambientCoefficient;
private final double diffuseCoefficient;
private final double specularCoefficient;
private final double reflectivity;
private final double ior;

// Constants
private static final double OPAQUE = 0.0;
private static final double TRANSPARENT = 1.0;
private static final Color TRANSPARENT_COLOR = new Color(0, 0, 0,
0);

private double transparency = OPAQUE;
private Matrix4 objectInverseTransform;

/***
 * Full constructor for ImageTextureMaterial.
 */
public ImageTextureMaterial(
    BufferedImage image,
    double uScale,
    double vScale,
    double uOffset,
    double vOffset,
    double ambientCoefficient,
    double diffuseCoefficient,
    double specularCoefficient,
    double shininess,
    double reflectivity,
    double ior,
    Matrix4 objectInverseTransform) {

    this.image = image;//ImageUtils3D.convertToTransparentImage(image,
this.transparency);
    this.uScale = uScale;
    this.vScale = vScale;
    this.uOffset = uOffset;
```

```

this.vOffset = vOffset;
this.ambientCoefficient = ambientCoefficient;
this.diffuseCoefficient = diffuseCoefficient;
this.specularCoefficient = specularCoefficient;
this.shininess = shininess;
this.reflectivity = clamp01(reflectivity);
this.ior = Math.max(1.0, ior);
this.objectInverseTransform = objectInverseTransform;
this.specularColor = Color.WHITE;
}

/**
 * Constructor for non-reflective and non-transparent textures.
 */
public ImageTextureMaterial(
    BufferedImage image,
    double uScale,
    double vScale,
    double uOffset,
    double vOffset,
    double ambientCoefficient,
    double diffuseCoefficient,
    double specularCoefficient,
    double shininess,
    Matrix4 objectInverseTransform) {

    this(image, uScale, vScale, uOffset, vOffset,
        ambientCoefficient, diffuseCoefficient, specularCoefficient,
        shininess, 0.0, 1.0, objectInverseTransform);
}

/**
 * Simplified constructor with default Phong properties.
 */
public ImageTextureMaterial(BufferedImage image, Matrix4
objectInverseTransform) {
    this(image, 1.0, 1.0, 0.0, 0.0,
        0.1, 0.7, 0.7, 32.0, objectInverseTransform);
}

```

```

/**
 * Simplified constructor with custom scale.
 */
public ImageTextureMaterial(BufferedImage image, double scale,
Matrix4 objectInverseTransform) {
    this(image, scale, scale, 0.0, 0.0,
        0.1, 0.7, 0.7, 32.0, objectInverseTransform);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        setTransparency (TRANSPARENT);
        return TRANSPARENT_COLOR;
    }

    // Transform to local coordinates
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    if (localNormal == null) {
        setTransparency (TRANSPARENT);
        return TRANSPARENT_COLOR;
    }

    // Get texture color with improved sampling
    Color textureColor = getTextureColor(localPoint, localNormal);
}

```

```

if (textureColor.getAlpha() < 6) {
    setTransparency(TRANSPARENT);
    return TRANSPARENT_COLOR;
} else {
    setTransparency(OPAQUE);
}

// Get light properties using LightProperties utility
LightProperties lightProps = LightProperties.getLightProperties(light,
worldPoint);

// Calculate lighting using ColorUtil for operations
return calculateLighting(textureColor, worldNormal, lightProps,
viewerPos, worldPoint);
}

/**
 * Calculates the final lighting contribution using Phong model.
 */
private Color calculateLighting(Color textureColor, Vector3
worldNormal,
LightProperties lightProps, Point3 viewerPos, Point3 worldPoint) {

    // Convert to float components for calculations
    float[] texRGB = ColorUtil.getFloatComponents(textureColor);
    float[] lightRGB = ColorUtil.getFloatComponents(lightProps.color);

    // Initialize result components
    float r = 0.0f;
    float g = 0.0f;
    float b = 0.0f;

    // Ambient component
    if (lightProps.direction.lengthSquared() == 0) { // Ambient light
        r = (float)(texRGB[0] * ambientCoefficient * lightProps.intensity *
lightRGB[0]);
        g = (float)(texRGB[1] * ambientCoefficient * lightProps.intensity *
lightRGB[1]);
        b = (float)(texRGB[2] * ambientCoefficient * lightProps.intensity *

```

```

lightRGB[2]);
} else {
// Diffuse component
double diffuseFactor = Math.max(0,
worldNormal.dot(lightProps.direction));

r = (float)(texRGB[0] * diffuseCoefficient * diffuseFactor *
lightRGB[0] * lightProps.intensity);
g = (float)(texRGB[1] * diffuseCoefficient * diffuseFactor *
lightRGB[1] * lightProps.intensity);
b = (float)(texRGB[2] * diffuseCoefficient * diffuseFactor *
lightRGB[2] * lightProps.intensity);

// Specular component
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir =
lightProps.direction.negate().reflect(worldNormal);
double specularFactor = Math.pow(Math.max(0,
reflectDir.dot(viewDir)), shininess);

float[] specRGB = ColorUtil.getFloatComponents(specularColor);

r += (float)(specRGB[0] * specularCoefficient * specularFactor *
lightRGB[0] * lightProps.intensity);
g += (float)(specRGB[1] * specularCoefficient * specularFactor *
lightRGB[1] * lightProps.intensity);
b += (float)(specRGB[2] * specularCoefficient * specularFactor *
lightRGB[2] * lightProps.intensity);
}

float origAlfa = (float)(textureColor.getAlpha ());

// Clamp and return final color
return new Color(
ColorUtil.clamp(r, 0.0f, 1.0f),
ColorUtil.clamp(g, 0.0f, 1.0f),
ColorUtil.clamp(b, 0.0f, 1.0f),
ColorUtil.clamp(origAlfa / 255F, 0.0f, 1.0f)
);

```

```

}

/***
 * Retrieves texture color with improved sampling to eliminate black
lines.
 */
private Color getTextureColor(Point3 localPoint, Vector3 localNormal) {
    if (image == null || localNormal == null) {
        return TRANSPARENT_COLOR;
    }

    final int imgWidth = image.getWidth();
    final int imgHeight = image.getHeight();

    if (imgWidth == 0 || imgHeight == 0) {
        return TRANSPARENT_COLOR;
    }

    // Calculate UV coordinates based on dominant normal axis
    double[] uv = calculateUVCoordinates(localPoint, localNormal);
    double u = uv[0];
    double v = uv[1];

    // Apply texture transformations with proper wrapping
    u = ((u * uScale) + uOffset) % 1.0;
    v = ((v * vScale) + vOffset) % 1.0;

    // Ensure positive coordinates
    if (u < 0) u += 1.0;
    if (v < 0) v += 1.0;

    // Flip V coordinate for image coordinate system
    v = 1.0 - v;

    // Use bilinear filtering to eliminate black lines
    return sampleTextureWithFiltering(u, v, imgWidth, imgHeight);
}

/***

```

```

* Calculates UV coordinates based on dominant normal axis.
*/
private double[] calculateUVCoordinates(Point3 localPoint, Vector3
localNormal) {
    double absX = Math.abs(localNormal.x);
    double absY = Math.abs(localNormal.y);
    double absZ = Math.abs(localNormal.z);

    double u, v;

    if (absX >= absY && absX >= absZ) {
        u = localPoint.y;
        v = localPoint.z;
    } else if (absY >= absX && absY >= absZ) {
        u = localPoint.x;
        v = localPoint.z;
    } else {
        u = localPoint.x;
        v = localPoint.y;
    }

    return new double[]{u, v};
}

```

```

/**
 * Samples texture with bilinear filtering to eliminate black lines.
*/
private Color sampleTextureWithFiltering(double u, double v, int
imgWidth, int imgHeight) {
    // Convert to pixel coordinates with sub-pixel precision
    double x = u * (imgWidth - 1);
    double y = v * (imgHeight - 1);

    // Get surrounding pixels for bilinear filtering
    int x0 = (int) Math.floor(x);
    int y0 = (int) Math.floor(y);
    int x1 = (int) Math.ceil(x);
    int y1 = (int) Math.ceil(y);

```

```

// Ensure coordinates are within bounds with proper wrapping
x0 = wrapCoordinate(x0, imgWidth);
y0 = wrapCoordinate(y0, imgHeight);
x1 = wrapCoordinate(x1, imgWidth);
y1 = wrapCoordinate(y1, imgHeight);

// Get colors of surrounding pixels
Color c00 = new Color(image.getRGB(x0, y0), true);
Color c10 = new Color(image.getRGB(x1, y0), true);
Color c01 = new Color(image.getRGB(x0, y1), true);
Color c11 = new Color(image.getRGB(x1, y1), true);

// Calculate interpolation factors
double tx = x - x0;
double ty = y - y0;

// Perform bilinear interpolation using ColorUtil
return ColorUtil.bilinearInterpolate(c00, c10, c01, c11, tx, ty);
}

/**
 * Wraps coordinate to stay within texture bounds.
 */
private int wrapCoordinate(int coord, int max) {
    if (coord < 0) return max - 1 - ((-coord - 1) % max);
    return coord % max;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override

```

```
public double getTransparency() {
    return transparency;
}

public void setTransparency(double transparency) {
    this.transparency = clamp01(transparency);
}

/**
 * Clamps value between 0.0 and 1.0.
 */
private double clamp01(double value) {
    return Math.max(0.0, Math.min(1.0, value));
}

}

// =====
// File: /net/elenamurat/material/DewDropMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

/**
 * Optimized dew drop material
 */
public class DewDropMaterial implements Material {
    private final Color baseColor;
    private final Color dropColor;
    private final double dropDensity; // [0.1, 0.9]
    private final double dropSize; // [0.01, 0.1]
    private Matrix4 objectInverseTransform;
```

```

private final double ambientCoefficient;
private final double diffuseCoefficient;
private final double specularCoefficient;
private final double shininess;
private final double reflectivity;
private final double ior; // 1.33 (water)
private final double transparency;

public DewDropMaterial(Color baseColor, Color dropColor,
    double dropDensity, double dropSize,
    double ambient, double diffuse, double specular,
    double shininess, double reflectivity,
    double ior, double transparency,
    Matrix4 objectInverseTransform) {
    this.baseColor = baseColor;
    this.dropColor = dropColor;
    this.dropDensity = clamp(dropDensity, 0.1, 0.9);
    this.dropSize = clamp(dropSize, 0.01, 0.1);
    this.ambientCoefficient = clamp(ambient);
    this.diffuseCoefficient = clamp(diffuse);
    this.specularCoefficient = clamp(specular);
    this.shininess = Math.max(1, shininess);
    this.reflectivity = clamp(reflectivity);
    this.ior = ior;
    this.transparency = clamp(transparency);
    this.objectInverseTransform = objectInverseTransform;
}

```

```

// Simple constructor
public DewDropMaterial(Color baseColor, Color dropColor,
    double dropDensity, double dropSize,
    Matrix4 objectInverseTransform) {
    this(baseColor, dropColor, dropDensity, dropSize,
        0.15, 0.6, 0.25, 50.0, 0.1, 1.33, 0.3,
        objectInverseTransform);
}

```

@Override

```

public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) return Color.BLACK;

    // Transform to object space
    Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
    Color finalColor = calculateDewColor(baseColor, localPoint,
worldNormal);

    // Light calculations
    Color lightColor = light.getColor();
    double intensity = light.getIntensityAt(worldPoint);
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

    // Phong lighting components
    int r = calculateAmbient(finalColor, lightColor);
    int g = r, b = r; // Same for all channels

    double NdotL = Math.max(0, worldNormal.dot(lightDir));
    if (NdotL > 0) {
        int[] diffuse = calculateDiffuse(finalColor, lightColor, intensity,
NdotL);
        int[] specular = calculateSpecular(lightColor, intensity,
lightDir, worldNormal, viewerPos);

        r = Math.min(255, r + diffuse[0] + specular[0]);
        g = Math.min(255, r + diffuse[1] + specular[1]);
        b = Math.min(255, r + diffuse[2] + specular[2]);
    }

    return new Color(r, g, b);
}

```

```

private Color calculateDewColor(Color base, Point3 localPoint, Vector3
normal) {
    if (!hasDewDrop(localPoint)) return base;

    double dist = calculateDropDistance(localPoint);
    double dropFactor = Math.max(0, 1 - (dist / dropSize));
    return ColorUtil.blendColors(base, dropColor, dropFactor * 0.8);
}

private boolean hasDewDrop(Point3 point) {
    Vector3 normal = point.toVector3().normalize();
    double noise = simpleNoise(normal.x * 100, normal.y * 100, normal.z *
100);
    return noise > (1.0 - dropDensity);
}

private double calculateDropDistance(Point3 point) {
    Vector3 normal = point.toVector3().normalize();
    return Math.sqrt(
        Math.pow(normal.x % (dropSize * 10), 2) +
        Math.pow(normal.y % (dropSize * 10), 2)
    );
}

private int calculateAmbient(Color color, Color lightColor) {
    return (int)(color.getRed() * ambientCoefficient *
(lightColor.getRed()/255.0));
}

private int[] calculateDiffuse(Color color, Color lightColor,
double intensity, double NdotL) {
    return new int[]{
        (int)(color.getRed() * diffuseCoefficient * NdotL *
(lightColor.getRed()/255.0) * intensity),
        (int)(color.getGreen() * diffuseCoefficient * NdotL *
(lightColor.getGreen()/255.0) * intensity),
        (int)(color.getBlue() * diffuseCoefficient * NdotL *
(lightColor.getBlue()/255.0) * intensity)
    };
}

```

```

    };
}

private int[] calculateSpecular(Color lightColor, double intensity,
    Vector3 lightDir, Vector3 normal, Point3 viewPos) {
    Vector3 viewDir = viewPos.subtract(viewPos).normalize();

    Vector3 reflectDir = lightDir.negate().reflect(normal);

    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specular = Math.pow(RdotV, shininess * 1.5) *
specularCoefficient * intensity;

    return new int[]{
        (int)(lightColor.getRed() * specular),
        (int)(lightColor.getGreen() * specular),
        (int)(lightColor.getBlue() * specular)
    };
}

// Helper methods
private double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

private double clamp(double value) {
    return clamp(value, 0, 1);
}

private double simpleNoise(double x, double y, double z) {
    // Simple hash function
    int hash = (int)(x * 127 + y * 311 + z * 571);
    return (hash & 0xFFFFFFFF) / 2147483647.0;
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }
}

```

```
/***
DewDropMaterial dewMat = new DewDropMaterial(
new Color(50, 120, 50), // Leaf color
new Color(200, 230, 255, 150), // Semi-transparent dew color
0.7, // Density (70%)
0.03, // Droplet size
sphere.getInverseTransform()
);
sphere.setMaterial(dewMat);
```

```
DewDropMaterial heavyDew = new DewDropMaterial(
Color.GRAY, // Rock color
new Color(220, 240, 255),
0.9, // Very dense
0.05, // Large droplets
object.getInverseTransform()
);
```

```
DewDropMaterial mistyDew = new DewDropMaterial(
new Color(70, 90, 110),
new Color(255, 255, 255, 180),
0.5,
0.02,
0.2, 0.5, 0.3, // Ambient, diffuse, specular
80.0, 0.15, 1.4, 0.4, // Shininess, reflectivity, IOR, transparency
rock.getInverseTransform()
);
*/
// =====
```

```
// File: /net/elenamurat/material/SolidColorMaterial.java
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

/**
 * SolidColorMaterial represents a simple material that returns a solid
color for any point,
 * without complex lighting calculations or patterns.
 * It now fully implements the extended Material interface.
*/
public class SolidColorMaterial implements Material {
    private final Color color;

    // Default values for new interface methods, as this is a simple solid color
material
    private final double reflectivity = 0.0;
    private final double ior = 1.0; // Index of Refraction for air/vacuum
    private final double transparency;// = 0.0;

    /**
     * Constructs a SolidColorMaterial with a specified color.
     * @param color The solid color of the material.
     */
    public SolidColorMaterial(Color color) {
        this.color = color;

        int alfa = (this.color).getAlpha ();
        double alpha = ((double)(alfa))/255.0;

        this.transparency = (1.0-alpha);
    }

    /**
     * Calculates the color at a given point on the surface. For
SolidColorMaterial,
     * it simply returns the predefined solid color, ignoring lighting and
viewer position.
     * @param point The point in 3D space (world coordinates).
     * @param normal The normal vector at the point (world coordinates).
     * @param light The light source (ignored for solid color).

```

```
* @param viewerPos The position of the viewer/camera (ignored for
solid color).
* @return The solid color of the material.
*/
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    return color; // SolidColor always returns the same color, no lighting
effect
}

/***
 * Returns the reflectivity coefficient. For a solid color material, this is
typically 0.0.
* @return 0.0
*/
@Override
public double getReflectivity() {
    return reflectivity;
}

/***
 * Returns the index of refraction. For an opaque solid color material, this
is typically 1.0.
* @return 1.0
*/
@Override
public double getIndexOfRefraction() {
    return ior;
}

/***
 * Returns the transparency coefficient. For an opaque solid color
material, this is typically 0.0.
* @return 0.0
*/
@Override
public double getTransparency() {
    return transparency;
}
```

```

}

@Override
public void setObjectTransform(Matrix4 tm) {
}

}

// =====
// File: /net/elenamurat/material/AnodizedTextMaterial.java
// =====

package net.elena.murat.material;

import java.awt.*;
import java.awt.geom.Point2D;
import java.awt.image.BufferedImage;
import java.util.Random;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

/**
 * AnodizedTextMaterial - Combines anodized metal material with
text/image texture support.
 * Renders text or images on an iridescent, metallic anodized surface.
 */
public class AnodizedTextMaterial implements Material {

    // --- TEXTURE PROPERTIES (from TextDielectricMaterial) ---
    private final String word;
    private final Color textColor;
    private final Color gradientColor;
    private final String gradientType;
    private final Color bgColor;
    private final String fontFamily;
    private final int fontStyle;
}

```

```

private final int fontSize;
private final int uOffset;
private final int vOffset;
private final BufferedImage imageObject;
private final int imageWidth;
private final int imageHeight;
private final int imageUOffset;
private final int imageVOffset;
private final BufferedImage texture;

// --- ANODIZED METAL PROPERTIES (from
AnodizedMetalMaterial) ---
private final Color baseColor;
private Matrix4 objectTransform;

// Phong parameters optimized for metallic surface
private final double ambientCoeff = 0.3;
private final double diffuseCoeff = 0.2; // Weak diffuse for metallic
private final double specularCoeff = 1.0; // Strong specular
private final double shininess = 100.0;
private final Color specularColor = Color.WHITE;
private final double reflectivity = 0.8;
private final double ior = 2.4;
private final double transparency = 0.0;

/**
 * Full constructor with all text and anodized properties
 */
public AnodizedTextMaterial(String word, Color textColor, Color
gradientColor,
String gradientType, Color bgColor,
String fontFamily, int fontStyle, int fontSize,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset,
Color baseColor) {

    // Text properties
    this.word = convertToNorwegianText(word).replaceAll("_", " ");
}

```

```

this.textColor = textColor;
this.gradientColor = gradientColor;
this.gradientType = gradientType != null ? gradientType : "horizontal";
this.bgColor = bgColor;
this.fontFamily = fontFamily.replaceAll("_", " ");
this.fontSize = fontSize;
this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

// Anodized properties
this.baseColor = baseColor != null ? baseColor : new Color(50, 50,
200);
this.objectTransform = Matrix4.identity();

// Generate texture
this.texture = createTexture();
}

/***
 * Simplified constructor with defaults
 */
public AnodizedTextMaterial(String word, Color textColor, String
fontFamily, int fontStyle, int fontSize) {
    this(word, textColor, null, "horizontal", new Color(0, 0, 0, 0),
        fontFamily, fontStyle, fontSize, 0, 0,
        null, 0, 0, 0, 0,
        new Color(50, 50, 200));
}

/***
 * Creates the texture image with the word drawn centered, optionally
with a gradient and background image.
* The texture size is fixed at 1024x1024 pixels.

```

```
*/  
private BufferedImage createTexture() {  
    final int size = 1024;  
    BufferedImage texture = new BufferedImage(size, size,  
BufferedImage.TYPE_INT_ARGB);  
    Graphics2D g2d = texture.createGraphics();  
  
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  
RenderingHints.VALUE_ANTIALIAS_ON);  
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,  
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);  
  
    g2d.setBackground(new Color(0, 0, 0, 0));  
    g2d.clearRect(0, 0, size, size);  
  
    // Optional background image  
    if (imageObject != null) {  
        int imgX = ((size - imageWidth) / 2) + imageUOffset;  
        int imgY = ((size - imageHeight) / 2) + imageVOffset;  
        g2d.drawImage(imageObject, imgX, imgY, imageWidth, imageHeight,  
null);  
    }  
  
    // Font setup  
    Font font;  
    try {  
        font = new Font(fontFamily, fontStyle, fontSize);  
    } catch (Exception e) {  
        font = new Font("Arial", fontStyle, fontSize); // Fallback  
    }  
    g2d.setFont(font);  
  
    FontMetrics fm = g2d.getFontMetrics();  
    int textWidth = fm.stringWidth(word);  
    int textHeight = fm.getHeight();  
    int ascent = fm.getAscent();  
  
    int x = ((size - textWidth) / 2) + uOffset;  
    int y = ((size - textHeight) / 2) + (ascent * 2) + (textHeight / 3) +
```

```

vOffset;

// Apply gradient or solid color
if (gradientColor != null) {
    GradientPaint gradient = createGradient(x, y - ascent, textWidth,
textHeight);
    g2d.setPaint(gradient);
} else {
    g2d.setColor(textColor);
}

g2d.drawString(word, x, y);
g2d.dispose();

return texture;
}

private GradientPaint createGradient(float x, float y, float width, float
height) {
    switch (gradientType.toLowerCase()) {
        case "vertical":
            return new GradientPaint(x, y, textColor, x, y + height / 2,
gradientColor, true);
        case "diagonal":
            return new GradientPaint(x, y, textColor, x + width / 3, y + height / 5,
gradientColor, true);
        case "horizontal":
        default:
            return new GradientPaint(x, y, textColor, x + width / 3, y,
gradientColor, true);
    }
}
}

public static String convertToNorwegianText(String input) {
    if (input == null || input.isEmpty()) return input;
    String result = input;
    result = result.replace("AE", "\u00C6");
    result = result.replace("O/", "\u00D8");
    result = result.replace("A0", "\u00C5");
}

```

```

result = result.replace("ae", "\u00E6");
result = result.replace("o/", "\u00F8");
result = result.replace("a0", "\u00E5");
return result;
}

// --- ANODIZED RENDERING WITH TEXTURE ---
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // 1. Get texture color at this point
    Point3 localPoint =
objectTransform.inverse().transformPoint(worldPoint);
    Color textureColor = getTextureColor(localPoint, worldNormal);

    // 2. If texture has alpha=0, use base anodized color only
    if (textureColor.getAlpha() == 0) {
        textureColor = baseColor;
    }

    // 3. Apply iridescence effect to the combined color
    Color surfaceColor = calculateIridescentColor(worldPoint,
worldNormal, viewerPos, textureColor);

    // 4. Handle lighting (same as AnodizedMetalMaterial)
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);
}

```

```

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

/***
 * Modified iridescence that blends baseColor with textureColor based on
view angle
*/
private Color calculateIridescentColor(Point3 worldPoint, Vector3
normal, Point3 viewerPos, Color textureColor) {
    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    double viewAngle = Math.abs(viewDir.dot(normal));

    // Extract RGB from texture
    int r = textureColor.getRed();
    int g = textureColor.getGreen();
    int b = textureColor.getBlue();

    // Blend with iridescent shift based on view angle
    if (viewAngle < 0.3) {
        // Narrow angle - blue shift
        r = (int)(r * 0.7);
        g = (int)(g * 0.8);
        b = (int)(b * 1.2);
    } else if (viewAngle < 0.6) {
        // Medium angle - purple shift
        r = (int)(r * 1.1);
        g = (int)(g * 0.7);
        b = (int)(b * 1.0);
    } else {
        // Wide angle - pink/gold shift
        r = (int)(r * 1.3);
    }
}

```

```

        g = (int)(g * 0.9);
        b = (int)(b * 0.8);
    }

    return new Color(
        Math.min(255, Math.max(0, r)),
        Math.min(255, Math.max(0, g)),
        Math.min(255, Math.max(0, b)),
        textureColor.getAlpha() // Preserve alpha
    );
}

private Color getTextureColor(Point3 localPoint, Vector3 worldNormal)
{
    if(texture == null) return new Color(0, 0, 0, 0);

    Vector3 dir = worldNormal.normalize();
    double phi = Math.atan2(dir.z, dir.x);
    double theta = Math.asin(dir.y);

    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI);
    double v = (theta + Math.PI / 2) / Math.PI;
    v = 1.0 - v;

    u = (u + 0.25) % 1.0; // Offset for better alignment

    int texX = (int) (u * texture.getWidth());
    texX = texX % texture.getWidth();
    if(texX < 0) texX += texture.getWidth();

    int texY = (int) (v * texture.getHeight());
    if(texY < 0 || texY >= texture.getHeight()) {
        return new Color(0, 0, 0, 0);
    }

    return new Color(texture.getRGB(texX, texY), true);
}

// --- MATERIAL INTERFACE ---

```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    this.objectTransform = (tm != null) ? tm : new Matrix4();
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

// --- GETTERS ---
public Color getBaseColor() { return baseColor; }
public BufferedImage getTexture() { return texture; }

@Override
public String toString() {
    return String.format("AnodizedTextMaterial[text='%s', baseColor=%s]", word, baseColor);
}

}

// =====
// File: /net/elenamurat/material/SilverMaterial.java
// =====

package net.elena.murat.material;
```

```
import java.awt.Color;

public class SilverMaterial extends MetallicMaterial {
    public SilverMaterial() {
        super(new Color(192, 192, 192), // Silver base color
              Color.WHITE, // White specular highlight for silver
              0.95, // Very high reflectivity
              200.0, // Very high shininess
              0.1, // Ambient light contribution
              0.02, // Very low diffuse contribution
              0.98 // Very high specular contribution
        );
    }
}
```

```
// =====
// File: /net/elenamurat/material/PureWaterMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

public class PureWaterMaterial implements Material {
    private final Color baseColor;
    private final double flowSpeed;
    private final long startTime;

    public PureWaterMaterial() {
        this(new Color(135, 206, 250), 0.1); // Default: Light blue, medium
        speed
    }

    public PureWaterMaterial(Color baseColor, double flowSpeed) {
```

```

        this.baseColor = baseColor != null ? baseColor : new Color(135, 206,
250);
        this.flowSpeed = Math.max(0.01, Math.min(1.0, flowSpeed));
        this.startTime = System.currentTimeMillis();
    }

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Time-based animation
    double time = (System.currentTimeMillis() - startTime) * 0.001 *
flowSpeed;

    // 2. Vertical wave pattern
    double verticalWave = bound(Math.sin(point.y * 12 + time * 1.5) *
0.25, -0.25, 0.25);

    // 3. Cross-wave turbulence
    double turbulence = Math.sin(point.x * 7 + time * 1.8) *
Math.cos(point.z * 5 - time * 2.2) *
0.15;

    // 4. Dynamic alpha
    double baseAlpha = 0.3 + 0.5 * (1 - bound(point.y, 0.1, 0.9));
    double alpha = bound(baseAlpha + verticalWave + turbulence, 0.15,
0.85);

    // 5. Pure water color (no foam)
    return new Color(
        clamp(baseColor.getRed() * (0.7 + turbulence * 0.3)),
        clamp(baseColor.getGreen() * (0.8 + turbulence * 0.2)),
        clamp(baseColor.getBlue() * (0.9 + turbulence * 0.1)),
        (int)(alpha * 255)
    );
}

@Override
public double getReflectivity() {
    return 0.2; // Slight reflection
}

```

```

}

@Override
public double getIndexOfRefraction() {
    return 1.33; // Water's refractive index
}

@Override
public double getTransparency() {
    return 0.7; // Semi-transparent
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

// Helper methods
private int clamp(double value) {
    return (int) Math.max(0, Math.min(255, value));
}

private double bound(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

}

/**
EMShape calmPond = new Plane()
.setMaterial(new PureWaterMaterial(
new Color(180, 220, 255), // Light blue
0.02 // Very slow flow
));

EMShape slowRiver = new Plane()
.setTransform(Matrix4.rotateX(Math.toRadians(10)))
.setMaterial(new PureWaterMaterial(
new Color(120, 190, 255), // Medium blue
0.08 // Slow flow
)

```

```
)  
*/  
  
// ======  
// File: /net/elenamurat/material/TextDielectricMaterial.java  
// ======
```

```
package net.elena.murat.material;
```

```
import java.awt.*;  
import java.awt.geom.Point2D;  
import java.awt.image.BufferedImage;  
import java.util.Random;
```

```
import net.elena.murat.math.*;  
import net.elena.murat.light.Light;  
import net.elena.murat.util.ColorUtil;
```

```
/**  
 * TextDielectricMaterial - Combines text rendering capability with  
dielectric material properties  
 * Creates transparent glass-like text on a sphere with refraction and  
reflection effects  
 */  
public class TextDielectricMaterial implements Material {
```

```
// Text properties (from SphereWordTextureMaterial)  
private final String word;  
private final Color textColor;  
private final Color gradientColor;  
private final String gradientType;  
private final Color bgColor;  
private final String fontFamily;  
private final int fontStyle;  
private final int fontSize;  
private final int uOffset;  
private final int vOffset;  
private final BufferedImage imageObject;
```

```

private final int imageWidth;
private final int imageHeight;
private final int imageUOffset;
private final int imageVOffset;
private BufferedImage texture;

// Dielectric properties (from DielectricMaterial)
private Color diffuseColor;
private double indexOfRefraction;
private double transparency;
private double reflectivity;
private Color filterColorInside;
private Color filterColorOutside;
private Matrix4 objectTransform;
private final Random random;
private double currentReflectivity;
private double currentTransparency;

/**
 * Constructor with text and dielectric properties
 */
public TextDielectricMaterial(String word, Color textColor, Color
gradientColor,
String gradientType, Color bgColor,
String fontFamily, int fontStyle, int fontSize,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset,
Color diffuseColor, double ior, double transparency, double reflectivity,
Color filterColorInside, Color filterColorOutside) {

    // Text properties
    this.word = convertToNorwegianText(word).replaceAll("_", " ");
    this.textColor = textColor;
    this.gradientColor = gradientColor;
    this.gradientType = gradientType != null ? gradientType : "horizontal";
    this.bgColor = bgColor;
    this.fontFamily = fontFamily.replaceAll("_", " ");
    this.fontStyle = fontStyle;
}

```

```

this.fontSize = fontSize;
this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

// Dielectric properties
this.diffuseColor = diffuseColor;
this.indexOfRefraction = ior;
this.transparency = transparency;
this.reflectivity = reflectivity;
this.filterColorInside = filterColorInside;
this.filterColorOutside = filterColorOutside;

this.random = new Random();
this.currentReflectivity = reflectivity;
this.currentTransparency = transparency;
this.objectTransform = new Matrix4().identity();

this.texture = createTexture();
}

/**
 * Simplified constructor with default dielectric properties
 */
public TextDielectricMaterial(String word, Color textColor,
    String fontFamily, int fontStyle, int fontSize) {
    this(word, textColor, null, "horizontal", new Color(0x00000000),
        fontFamily, fontStyle, fontSize, 0, 0,
        null, 0, 0, 0, 0,
        new Color(0.9f, 0.9f, 0.9f), 1.5, 0.8, 0.1,
        new Color(1.0f, 1.0f, 1.0f), new Color(1.0f, 1.0f, 1.0f));
}

/**
 * Creates the texture image with the word drawn centered, optionally

```

with a gradient and background image.

* The texture size is fixed at 1024x1024 pixels.

*

* @return BufferedImage containing the rendered word texture.

*/

```
private BufferedImage createTexture() {
```

```
    final int size = 1024;
```

```
    BufferedImage texture = new BufferedImage(size, size,  
    BufferedImage.TYPE_INT_ARGB);
```

```
    Graphics2D g2d = texture.createGraphics();
```

```
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,  
    RenderingHints.VALUE_ANTIALIAS_ON);
```

```
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,  
    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
```

```
    g2d.setBackground(new Color(0, 0, 0, 0));
```

```
    g2d.clearRect(0, 0, size, size);
```

```
    if (imageObject != null) {
```

```
        int imgX = ((size - imageWidth) / 2) + imageUOffset;
```

```
        int imgY = ((size - imageHeight) / 2) + imageVOffset;
```

```
        g2d.drawImage(imageObject, imgX, imgY, imageWidth, imageHeight,  
        null);
```

```
}
```

```
Font font;
```

```
try {
```

```
    font = new Font(fontFamily, fontStyle, fontSize);
```

```
} catch (Exception e) {
```

```
    font = new Font("Arial", fontStyle, fontSize); // Fallback font
```

```
}
```

```
g2d.setFont(font);
```

```
FontMetrics fm = g2d.getFontMetrics();
```

```
int textWidth = fm.stringWidth(word);
```

```
int textHeight = fm.getHeight();
```

```
int ascent = fm.getAscent();
```

```
int x = ((size - textWidth) / 2) + uOffset;
int y = ((size - textHeight) / 2) + (ascent * 2) + (textHeight / 3) +
vOffset;

if (gradientColor != null) {
    GradientPaint gradient = createGradient(x, y - ascent, textWidth,
textHeight);
    g2d.setPaint(gradient);
} else {
    g2d.setColor(textColor);
}

g2d.drawString(word, x, y);
g2d.dispose();

return texture;
}

private GradientPaint createGradient(float x, float y, float width, float
height) {
    switch (gradientType.toLowerCase()) {
        case "vertical":
            return new GradientPaint(x, y, textColor, x, y + height/2,
gradientColor, true);
        case "diagonal":
            return new GradientPaint(x, y, textColor, x + width/3, y + height/5,
gradientColor, true);
        case "horizontal":
        default:
            return new GradientPaint(x, y, textColor, x + width/3, y,
gradientColor, true);
    }
}

public static String convertToNorwegianText(String input) {
    if (input == null || input.isEmpty()) {
        return input;
    }
```

```

String result = input;
result = result.replace("AE", "\u00C6");
result = result.replace("O/", "\u00D8");
result = result.replace("A0", "\u00C5");
result = result.replace("ae", "\u00E6");
result = result.replace("o/", "\u00F8");
result = result.replace("a0", "\u00E5");

return result;
}

// Dielectric material methods
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    // Get texture color first
    Point3 localPoint = objectTransform.inverse().transformPoint(point);
    Color textureColor = getTextureColor(localPoint, normal);

    // Apply dielectric lighting effects
    Vector3 lightDir = light.getDirectionTo(point).normalize();
    double diffuseFactor = Math.max(0.3, normal.dot(lightDir));

    // Fresnel effect for dynamic properties
    Vector3 viewDir = viewerPoint.subtract(point).normalize();
    double fresnel = Vector3.calculateFresnel(viewDir, normal, 1.0,
indexOfRefraction);

    this.currentReflectivity = Math.min(0.95, reflectivity + (fresnel * 0.8));
    this.currentTransparency = Math.max(0.05, transparency * (1.0 - fresnel
* 0.2));

    // Diffuse and specular components
    //Original is below line with textureColor:
    Color diffuse = ColorUtil.multiplyColor(textureColor, diffuseFactor *
light.getIntensity());

    Vector3 reflectDir = lightDir.reflect(normal);
    double specularFactor = Math.pow(Math.max(0,

```

```

viewDir.dot(reflectDir)), 128);
    Color specular = ColorUtil.multiplyColor(light.getColor(),
specularFactor * 1.2 * light.getIntensity());

    // Combine with glass tint
    Color result = ColorUtil.add(diffuse, specular);
    //if (transparency > 0.3) {
    //Color glassTint = new Color(0.95f, 0.97f, 1.0f);
    //result = ColorUtil.multiplyColors(result, glassTint);
    //}

    return ColorUtil.clampColor(result);
}

private Color getTextureColor(Point3 localPoint, Vector3 worldNormal)
{
    if (texture == null) return textColor;

    // Normal normalize
    Vector3 dir = worldNormal.normalize();

    double phi = Math.atan2(dir.z, dir.x);
    double theta = Math.asin(dir.y);

    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI);
    double v = (theta + Math.PI / 2) / Math.PI;
    v = 1.0 - v;

    u = (u + 0.25) % 1.0; // Offset

    int texX = (int) (u * texture.getWidth());
    texX = texX % texture.getWidth();
    if (texX < 0) texX += texture.getWidth();

    int texY = (int) (v * texture.getHeight());
    if (texY < 0 || texY >= texture.getHeight()) {
        return new Color(0, 0, 0, 0);
    }
    return new Color(texture.getRGB(texX, texY), true);
}

```

```
}
```

```
// Material interface methods
```

```
@Override
```

```
public void setObjectTransform(Matrix4 tm) {
```

```
    this.objectTransform = (tm != null) ? tm : new Matrix4();
```

```
}
```

```
@Override
```

```
public double getIndexOfRefraction() {
```

```
    return indexOfRefraction;
```

```
}
```

```
@Override
```

```
public double getTransparency() {
```

```
    return currentTransparency;
```

```
}
```

```
@Override
```

```
public double getReflectivity() {
```

```
    return currentReflectivity;
```

```
}
```

```
// Getters and setters for dielectric properties
```

```
public Color getFilterColorInside() { return filterColorInside; }
```

```
public Color getFilterColorOutside() { return filterColorOutside; }
```

```
public void setFilterColorInside(Color filterInside) {
```

```
    this.filterColorInside = filterInside;
```

```
}
```

```
public void setFilterColorOutside(Color filterOutside) {
```

```
    this.filterColorOutside = filterOutside;
```

```
}
```

```
public Color getDiffuseColor() { return diffuseColor; }
```

```
public void setDiffuseColor(Color color) { this.diffuseColor = color; }
```

```
public void setIndexOfRefraction(double ior) { this.indexOfRefraction =
```

```
    ior; }

    public void setTransparency(double transparency) { this.transparency =
transparency; }

    public void setReflectivity(double reflectivity) { this.reflectivity =
reflectivity; }

    @Override
    public String toString() {
        return String.format("TextDielectricMaterial[text=%s, ior=%.2f,
transparency=%.2f, reflectivity=%.2f]",
word, indexOfRefraction, transparency, reflectivity);
    }

}
```

```
// =====
// File: /net/elenamurat/material/LightningMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;
import java.util.Random;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

public class LightningMaterial implements Material {
    private final Color baseColor;
    private final double intensity;
    private final Random random;
    private long lastStrikeTime;
    private double[][] lightningPath;

    public LightningMaterial() {
        this(new Color(135, 206, 250), 1.5); // Electric blue
    }
```

```

public LightningMaterial(Color baseColor, double intensity) {
    this.baseColor = baseColor;
    this.intensity = Math.max(0.5, Math.min(5.0, intensity));
    this.random = new Random();
    this.lastStrikeTime = System.currentTimeMillis();
    generateLightningPath();
}

private void generateLightningPath() {
    // Lichtenberg figure algorithm (fractal lightning)
    int segments = 50;
    lightningPath = new double[segments][3]; // x,y,z

    double x = 0, y = 1, z = 0; // Start from ceiling
    lightningPath[0] = new double[] {x, y, z};

    for (int i = 1; i < segments; i++) {
        // Random direction change (fractal branching)
        x += (random.nextDouble() - 0.5) * 0.3;
        y -= random.nextDouble() * 0.2; // Downward
        z += (random.nextDouble() - 0.5) * 0.1;

        lightningPath[i] = new double[] {x, y, z};
    }
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Lightning refresh check (at random intervals)
    long currentTime = System.currentTimeMillis();
    if (currentTime - lastStrikeTime > 2000 + random.nextInt(3000)) {
        generateLightningPath();
        lastStrikeTime = currentTime;
    }

    // 2. Find closest lightning segment
    double minDist = Double.MAX_VALUE;
    for (int i = 0; i < lightningPath.length - 1; i++) {

```

```

        double dist = distanceToLineSegment(
            point,
            new Point3(lightningPath[i][0], lightningPath[i][1], lightningPath[i]
[2]),
            new Point3(lightningPath[i+1][0], lightningPath[i+1][1],
lightningPath[i+1][2])
        );
        minDist = Math.min(minDist, dist);
    }

    // 3. Calculate brightness (inverse square law)
    double brightness = intensity / (1.0 + 100 * minDist * minDist);

    // 4. Flicker effect
    double flicker = 0.8 + 0.2 * Math.sin(currentTime * 0.05);

    return new Color(
        (int)(baseColor.getRed() * brightness * flicker),
        (int)(baseColor.getGreen() * brightness * flicker),
        (int)(baseColor.getBlue() * brightness * flicker),
        (int)(255 * Math.min(1, brightness * 2)) // Alpha
    );
}

private double distanceToLineSegment(Point3 p, Point3 a, Point3 b) {
    Vector3 ap = p.subtract(a);
    Vector3 ab = b.subtract(a);

    double projection = ap.dot(ab) / ab.dot(ab);
    projection = Math.max(0, Math.min(1, projection));

    Point3 closest = a.add(ab.scale(projection));
    return p.distance(closest);
}

@Override public double getReflectivity() { return 0.3; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.8; }

```

```

@Override
public void setObjectTransform(Matrix4 tm) {
}

/**
// 1. Create black ceiling
EMShape ceiling = new Plane(new Point3(0, 5, 0), new Vector3(0, -1, 0))
.setMaterial(new SolidColorMaterial(Color.BLACK));

// 2. Lightning material
LightningMaterial lightning = new LightningMaterial(
new Color(200, 230, 255), // Whiter lightning
2.5 // Intensity
);

// 3. Thin plane for visualization
EMShape lightningViz = new Plane()
.setTransform(Matrix4.scale(10, 10, 0.1).translate(0, 2.5, 0))
.setMaterial(lightning);

scene.add(ceiling);
scene.add(lightningViz);
*/

```

```

// =====
// File: /net/elenamurat/material/TriangleMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

//custom
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Matrix4;

```

```

import net.elena.murat.math.Ray;
import net.elena.murat.light.*;

/**
 * A material that creates a repeating triangular pattern on a surface.
 * It applies a Phong-like lighting model (Ambient + Diffuse + Specular)
to the pattern.
 * This material now fully implements the extended Material interface.
 */

public class TriangleMaterial implements Material {

    private final Color color1; // Color for the first type of triangle/area
    private final Color color2; // Color for the second type of triangle/area
    private final double triangleSize; // Controls the size/frequency of the
triangles
    private Matrix4 objectInverseTransform; // The inverse transformation
matrix of the object

    // Lighting coefficients (Phong model)
    private final double ambientCoefficient;
    private final double diffuseCoefficient;
    private final double specularCoefficient;
    private final double shininess;
    private final double reflectivity;
    private final double ior; // Index of Refraction
    private final double transparency;

    // Default specular color (can be made a constructor parameter if needed)
    private final Color specularColor = Color.WHITE;

    /**
     * Full constructor for TriangleMaterial.
     * @param color1 First color for the triangle pattern.
     * @param color2 Second color for the triangle pattern.
     * @param triangleSize Controls the scale of the triangular pattern.
Smaller values mean more, smaller triangles.
     * @param ambientCoefficient Ambient light contribution.
     * @param diffuseCoefficient Diffuse light contribution.
     * @param specularCoefficient Specular light contribution.

```

```

* @param shininess Shininess for specular highlights.
* @param reflectivity Material's reflectivity (0.0 to 1.0).
* @param ior Index of refraction for transparent materials.
* @param transparency Material's transparency (0.0 to 1.0).
* @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.
*/
public TriangleMaterial(Color color1, Color color2, double triangleSize,
double ambientCoefficient, double diffuseCoefficient,
double specularCoefficient, double shininess,
double reflectivity, double ior, double transparency,
Matrix4 objectInverseTransform) { // Added objectInverseTransform
this.color1 = color1;
this.color2 = color2;
this.triangleSize = triangleSize;
this.objectInverseTransform = objectInverseTransform; // Store the
inverse transform

this.ambientCoefficient = ambientCoefficient;
this.diffuseCoefficient = diffuseCoefficient;
this.specularCoefficient = specularCoefficient;
this.shininess = shininess;
this.reflectivity = reflectivity;
this.ior = ior;
this.transparency = transparency;
}

/***
* Simplified constructor with default Phong-like coefficients.
* @param color1 First color for the triangle pattern.
* @param color2 Second color for the triangle pattern.
* @param triangleSize Controls the scale of the triangular pattern.
* @param objectInverseTransform The full inverse transformation
matrix of the object.
*/
public TriangleMaterial(Color color1, Color color2, double triangleSize,
Matrix4 objectInverseTransform) { // Added objectInverseTransform
// Default Phong parameters, adjusted to be similar to a common
"diffuse" material.

```

```

this(color1, color2, triangleSize,
    0.1, // ambientCoefficient
    0.8, // diffuseCoefficient
    0.1, // specularCoefficient
    10.0, // shininess
    0.0, // reflectivity
    1.0, // indexOfRefraction
    0.0, // transparency
    objectInverseTransform // Pass the inverse transform
);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

/**
 * Calculates the base pattern color at a given UV coordinate pair.
 * This method separates the pattern generation logic from lighting
calculations.
 *
 * @param u U texture coordinate (can be any value, will be normalized
internally).
 * @param v V texture coordinate (can be any value, will be normalized
internally).
 * @return The pattern color (either color1 or color2).
 */
private Color getPatternColor(double u, double v) {
    // Apply triangleSize as an inverse scale: smaller triangleSize = more
repetitions (smaller triangles)
    // This scales the UV coordinates before the pattern logic.
    u = u / triangleSize;
    v = v / triangleSize;

    // Ensure positive values for modulo operations and make the pattern
repeat seamlessly.
    // This handles negative u/v values correctly for wrapping.
}

```

```

// Adding EPSILON to prevent floating point issues near integer
boundaries.
double u_normalized = (u % 1.0 + 1.0) % 1.0;
double v_normalized = (v % 1.0 + 1.0) % 1.0;

// Determine which repeating "square" (cell) we are in.
// These `checkX` and `checkY` are used to alternate the pattern's
orientation/colors.
int checkX = (int) Math.floor(u); // Integer part of u
int checkY = (int) Math.floor(v); // Integer part of v

// --- Triangular Pattern Logic ---
// This divides each conceptual "square" cell (from UV mapping) into
two triangles
// based on a diagonal. The colors are swapped in alternating cells to
create
// a continuous triangular grid appearance.

Color patternColor;
if ((checkX + checkY) % 2 == 0) { // For "even" cells (e.g., (0,0), (0,2),
(1,1) etc.)
    if (u_normalized + v_normalized < 1.0) { // Top-left triangle (above
the diagonal from bottom-left to top-right)
        patternColor = color1;
    } else { // Bottom-right triangle (below the diagonal)
        patternColor = color2;
    }
} else { // For "odd" cells (e.g., (0,1), (1,0), (1,2) etc.)
    if (u_normalized + v_normalized < 1.0) { // Top-left triangle (colors
swapped for contrast)
        patternColor = color2;
    } else { // Bottom-right triangle (colors swapped)
        patternColor = color1;
    }
}
return patternColor;
}

/***

```

- * Returns the final shaded color at a given 3D point on the surface,
- * applying the triangular pattern and a Phong-like lighting model.
- * Uses normal-based planar UV mapping for better generalization across shapes.

*

- * @param worldPoint The 3D point on the surface in world coordinates.
- * @param worldNormal The surface normal at that point in world coordinates (should be normalized).

* @param light The light source.

* @param viewerPos The position of the viewer/camera.

* @return The final shaded color.

*/

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light light, Point3 viewerPos) {
```

// Check if inverse transform is valid before proceeding

if (objectInverseTransform == null) {

System.err.println("Error: TriangleMaterial's inverse transform is null.
Returning black.");

return Color.BLACK;

}

```
// 1. Transform the intersection point from world space into the object's local space
```

```
// This is crucial for applying the pattern correctly regardless of object's position, rotation, or scale.
```

Point3 localPoint =

```
objectInverseTransform.transformPoint(worldPoint);
```

```
// 2. Transform the world normal to local space to determine the local face orientation
```

Vector3 localNormal =

```
objectInverseTransform.inverseTransposeForNormal().transformVector(worldNormal).normalize();
```

// Check if the transformed normal is valid

if (localNormal == null) {

System.err.println("Error: TriangleMaterial's normal transform matrix is null or invalid. Returning black.");

return Color.BLACK;

```
}
```

```
// --- UV Coordinate Calculation (Normal-based Planar Mapping) ---
```

```
// This approach selects a projection plane based on the dominant axis of  
the surface normal.
```

```
// It's a common technique for procedural textures that need to wrap  
around arbitrary shapes.
```

```
// The localPoint is used for UV calculation to ensure pattern is aligned  
with object's local axes.
```

```
double u, v;
```

```
double absNx = Math.abs(localNormal.x);
```

```
double absNy = Math.abs(localNormal.y);
```

```
double absNz = Math.abs(localNormal.z);
```

```
// Project the 3D local point onto a 2D plane based on the dominant  
local normal axis.
```

```
// The scaling factor will control the density/size of the pattern.
```

```
// A value of 1.0 means 1 unit in world space corresponds to 1 unit in  
UV space (before triangleSize).
```

```
// Add a small epsilon to the coordinates before flooring to handle  
floating point inaccuracies at boundaries.
```

```
if (absNx > absNy && absNx > absNz) { // Normal is mostly X-axis  
(local Y-Z plane)
```

```
    u = localPoint.y + Ray.EPSILON;
```

```
    v = localPoint.z + Ray.EPSILON;
```

```
} else if (absNy > absNx && absNy > absNz) { // Normal is mostly Y-  
axis (local X-Z plane)
```

```
    u = localPoint.x + Ray.EPSILON;
```

```
    v = localPoint.z + Ray.EPSILON;
```

```
} else { // Normal is mostly Z-axis (local X-Y plane) or equally  
dominant
```

```
    u = localPoint.x + Ray.EPSILON;
```

```
    v = localPoint.y + Ray.EPSILON;
```

```
}
```

```
// Get the base color from the procedural triangular pattern.
```

```
Color patternBaseColor = getPatternColor(u, v);
```

```

// --- Lighting Calculation (Ambient + Diffuse + Specular Phong Model)
---

// This part is consistent with your other material classes.

Color lightColor = light.getColor();
double attenuatedIntensity = 0.0; // Initialize attenuatedIntensity here

// Ambient component
int rAmbient = (int) (patternBaseColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
int gAmbient = (int) (patternBaseColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
int bAmbient = (int) (patternBaseColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);

if (light instanceof ElenaMuratAmbientLight) {
    return new Color(
        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

Vector3 lightDirection;
if (light instanceof MuratPointLight) {
    MuratPointLight pLight = (MuratPointLight) light;
    lightDirection = pLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = pLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof ElenaDirectionalLight) {
    ElenaDirectionalLight dLight = (ElenaDirectionalLight) light;
    lightDirection = dLight.getDirection().negate().normalize();
    attenuatedIntensity = dLight.getIntensity();
} else if (light instanceof PulsatingPointLight) {
    PulsatingPointLight ppLight = (PulsatingPointLight) light;
    lightDirection =
ppLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = ppLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof SpotLight) {

```

```

SpotLight sLight = (SpotLight) light;
lightDirection = sLight.getDirectionAt(worldPoint);
attenuatedIntensity = sLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BioluminescentLight) {
BioluminescentLight bLight = (BioluminescentLight) light;
lightDirection = bLight.getDirectionAt(worldPoint);
attenuatedIntensity = bLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BlackHoleLight) {
BlackHoleLight bhLight = (BlackHoleLight) light;
lightDirection = bhLight.getDirectionAt(worldPoint);
attenuatedIntensity = bhLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof FractalLight) {
FractalLight fLight = (FractalLight) light;
lightDirection = fLight.getDirectionAt(worldPoint);
attenuatedIntensity = fLight.getAttenuatedIntensity(worldPoint);
} else {
System.err.println("Warning: Unknown or unsupported light type for
TriangleMaterial shading: " + light.getClass().getName());
return Color.BLACK;
}

```

```

// Diffuse component
double NdotL = Math.max(0, worldNormal.dot(lightDirection));
int rDiffuse = (int) (patternBaseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int) (patternBaseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int) (patternBaseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

```

```

// Specular component
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectionVector =
lightDirection.negate().reflect(worldNormal); // Use negate() before
reflect() for correct reflection vector
double RdotV = Math.max(0, reflectionVector.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

```

```

int rSpecular = (int) (specularColor.getRed() * specularCoefficient *

```

```

lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
    int gSpecular = (int) (specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
    int bSpecular = (int) (specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

    // Sum up all components for this light source
    // Ambient is handled separately for ambient lights. For other lights,
combine diffuse and specular.
    int finalR = Math.min(255, rDiffuse + rSpecular);
    int finalG = Math.min(255, gDiffuse + gSpecular);
    int finalB = Math.min(255, bDiffuse + bSpecular);

    return new Color(finalR, finalG, finalB);
}

/***
 * Returns the reflectivity coefficient of the material.
 * @return The reflectivity value (0.0-1.0).
 */
@Override
public double getReflectivity() { return reflectivity; }

/***
 * Returns the index of refraction (IOR) of the material.
 * @return The index of refraction.
 */
@Override
public double getIndexOfRefraction() { return ior; }

/***
 * Returns the transparency coefficient of the material.
 * @return The transparency value (0.0-1.0).
 */
@Override
public double getTransparency() { return transparency; }

/***
 * Helper method to get the normalized light direction vector from a light

```

source to a point.

```
* @param light The light source.  
* @param worldPoint The point in world coordinates.  
* @return Normalized light direction vector, or null if light type is  
unsupported.  
*/  
private Vector3 getLightDirection(Light light, Point3 worldPoint) {  
    if (light instanceof MuratPointLight) {  
        return  
((MuratPointLight)light).getPosition().subtract(worldPoint).normalize();  
    } else if (light instanceof ElenaDirectionalLight) {  
        return  
((ElenaDirectionalLight)light).getDirection().negate().normalize();  
    } else if (light instanceof PulsatingPointLight) {  
        return  
((PulsatingPointLight)light).getPosition().subtract(worldPoint).normalize()  
;  
    } else if (light instanceof SpotLight) {  
        return ((SpotLight)light).getDirectionAt(worldPoint).normalize();  
    } else if (light instanceof BioluminescentLight) {  
        return  
((BioluminescentLight)light).getDirectionAt(worldPoint).normalize();  
    } else if (light instanceof BlackHoleLight) {  
        return  
((BlackHoleLight)light).getDirectionAt(worldPoint).normalize();  
    } else if (light instanceof FractalLight) {  
        return ((FractalLight)light).getDirectionAt(worldPoint).normalize();  
    } else {  
        System.err.println("Warning: Unknown light type in  
PlaneFaceElenaTextureMaterial");  
        return new Vector3(0, 1, 0); // Varsayılan yön  
    }  
}  
  
// ======  
// File: /net/elena/murat/material/NorthernLightMaterial.java
```

```
// ======

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class NorthernLightMaterial implements Material {

    private final Color primaryAurora;
    private final Color secondaryAurora;
    private final double intensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.2;
    private final double diffuseCoeff = 0.4;
    private final double specularCoeff = 0.9;
    private final double shininess = 80.0;
    private final double reflectivity = 0.3;
    private final double ior = 1.45;
    private final double transparency = 0.6;

    public NorthernLightMaterial() {
        this(new Color(0x00, 0xFF, 0x7F), new Color(0x00, 0xBF, 0xFF), 0.85);
    }

    public NorthernLightMaterial(Color primaryAurora, Color secondaryAurora, double intensity) {
        this.primaryAurora = primaryAurora;
        this.secondaryAurora = secondaryAurora;
        this.intensity = Math.max(0, Math.min(1, intensity));
        this.objectTransform = Matrix4.identity();
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
```

```

    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateAuroraEffect(objectPoint, worldNormal,
viewerPos);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(Color.WHITE, props.color,
specularCoeff * specFactor);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateAuroraEffect(Point3 point, Vector3 normal, Point3

```

```

viewerPos) {
    double x = point.x * 8.0;
    double y = point.y * 12.0;
    double z = point.z * 8.0;

    // Aurora curtain wave patterns
    double curtain1 = Math.sin(x * 0.7 + Math.sin(y * 1.2) * 3.0 + z * 0.3);
    double curtain2 = Math.cos(y * 1.5 + Math.sin(x * 0.9) * 2.5 + z * 0.4);
    double curtain3 = Math.sin(x * 1.1 + y * 2.0 + Math.cos(z * 0.6) * 2.0);

    double auroraPattern = (curtain1 * 0.5 + curtain2 * 0.3 + curtain3 * 0.2);
    double normalizedPattern = (auroraPattern + 1.0) * 0.5;

    // Time-based animation simulation (using z-coordinate as time proxy)
    double timeEffect = Math.sin(z * 0.5 + System.currentTimeMillis() *
0.0001) * 0.3 + 0.7;

    // View-dependent intensity
    Vector3 viewDir = viewerPos.subtract(point).normalize();
    double viewEffect = Math.pow(Math.abs(viewDir.dot(normal)), 0.3);

    double finalIntensity = intensity * timeEffect * viewEffect;

    if (normalizedPattern < 0.6) {
        // Primary aurora green
        double ratio = normalizedPattern / 0.6;
        Color baseAurora = ColorUtil.blendColors(primaryAurora,
secondaryAurora, ratio * 0.4);
        return ColorUtil.lightenColor(baseAurora, finalIntensity * 0.8);
    } else {
        // Secondary aurora blue with glow effect
        double ratio = (normalizedPattern - 0.6) / 0.4;
        Color glowingAurora = ColorUtil.blendColors(secondaryAurora,
primaryAurora, ratio * 0.2);

        // Add emission glow
        int r = glowingAurora.getRed() + (int)(finalIntensity * 50);
        int g = glowingAurora.getGreen() + (int)(finalIntensity * 40);
        int b = glowingAurora.getBlue() + (int)(finalIntensity * 30);
    }
}

```

```
        return ColorUtil.createColor(r, g, b);
    }
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/elenamurat/material/RectangleCheckerMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

/**
 * Material that creates a rectangular checkerboard pattern.
 * Rectangular version of the classic checkerboard.
 */
public class RectangleCheckerMaterial implements Material {
```

```
private final Color color1;
private final Color color2;
private final double rectWidth;
private final double rectHeight;
private Matrix4 objectInverseTransform;

// Lighting coefficients
private final double ambientCoefficient;
private final double diffuseCoefficient;
private final double specularCoefficient;
private final double shininess;
private final double reflectivity;
private final double ior;
private final double transparency;
private final Color specularColor;

/***
 * Full constructor
 * @param color1 First color
 * @param color2 Second color
 * @param rectWidth Dikdörtgen genişliği (tekrar aralığı)
 * @param rectHeight Dikdörtgen yüksekliği (tekrar aralığı)
 * @param ambient Ambient coefficient
 * @param diffuse Diffuse coefficient
 * @param specular Specular coefficient
 * @param shininess Shininess exponent
 * @param reflectivity Reflectivity amount (0-1)
 * @param ior Index of refraction
 * @param transparency Transparency amount (0-1)
 * @param objectInverseTransform Object's inverse transform
 */
public RectangleCheckerMaterial(Color color1, Color color2,
    double rectWidth, double rectHeight,
    double ambient, double diffuse, double specular,
    double shininess, double reflectivity,
    double ior, double transparency,
    Matrix4 objectInverseTransform) {
    this.color1 = color1;
    this.color2 = color2;
```

```

this.rectWidth = rectWidth;
this.rectHeight = rectHeight;
this.ambientCoefficient = ambient;
this.diffuseCoefficient = diffuse;
this.specularCoefficient = specular;
this.shininess = shininess;
this.reflectivity = reflectivity;
this.ior = ior;
this.transparency = transparency;
this.objectInverseTransform = objectInverseTransform;
this.specularColor = Color.WHITE;
}

/**
* Simplified constructor with default Phong parameters
*/
public RectangleCheckerMaterial(Color color1, Color color2,
    double rectWidth, double rectHeight,
    Matrix4 objectInverseTransform) {
    this(color1, color2, rectWidth, rectHeight,
        0.1, 0.7, 0.2, 10.0, 0.0, 1.0, 0.0,
        objectInverseTransform);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

/**
* Generates the checker pattern color
*/
private Color getPatternColor(double u, double v) {
    // Adjust for rectangle dimensions
    double uScaled = u / rectWidth;
    double vScaled = v / rectHeight;

    // Handle negative coordinates correctly
}

```

```

uScaled = (uScaled % 2.0 + 2.0) % 2.0;
vScaled = (vScaled % 2.0 + 2.0) % 2.0;

int xSeg = (int)Math.floor(uScaled);
int ySeg = (int)Math.floor(vScaled);

return (xSeg + ySeg) % 2 == 0 ? color1 : color2;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        return Color.BLACK;
    }

    // Transform to object space
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal().transformVector(w
    orldNormal).normalize();

    // Determine dominant axis for UV mapping
    double absX = Math.abs(localNormal.x);
    double absY = Math.abs(localNormal.y);
    double absZ = Math.abs(localNormal.z);

    double u, v;
    if (absX > absY && absX > absZ) {
        u = localPoint.y;
        v = localPoint.z;
    } else if (absY > absX && absY > absZ) {
        u = localPoint.x;
        v = localPoint.z;
    } else {
        u = localPoint.x;
        v = localPoint.y;
    }
}

```

```

// Get base pattern color
Color baseColor = getPatternColor(u, v);

// Lighting calculations (Phong model)
Color lightColor = light.getColor();
double attenuatedIntensity = 0.0;

// Ambient component
int rAmbient = (int)(baseColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
int gAmbient = (int)(baseColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
int bAmbient = (int)(baseColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);

if (light instanceof ElenaMuratAmbientLight) {
    return new Color(
        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
}

// Handle different light types
Vector3 lightDirection;
if (light instanceof MuratPointLight) {
    MuratPointLight pLight = (MuratPointLight) light;
    lightDirection = pLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = pLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof ElenaDirectionalLight) {
    ElenaDirectionalLight dLight = (ElenaDirectionalLight) light;
    lightDirection = dLight.getDirection().negate().normalize();
    attenuatedIntensity = dLight.getIntensity();
} else if (light instanceof PulsatingPointLight) {
    PulsatingPointLight ppLight = (PulsatingPointLight) light;
    lightDirection =
ppLight.getPosition().subtract(worldPoint).normalize();
    attenuatedIntensity = ppLight.getAttenuatedIntensity(worldPoint);
}

```

```

} else if (light instanceof SpotLight) {
    SpotLight sLight = (SpotLight) light;
    lightDirection = sLight.getDirectionAt(worldPoint);
    attenuatedIntensity = sLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BioluminescentLight) {
    BioluminescentLight bLight = (BioluminescentLight) light;
    lightDirection = bLight.getDirectionAt(worldPoint);
    attenuatedIntensity = bLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof BlackHoleLight) {
    BlackHoleLight bhLight = (BlackHoleLight) light;
    lightDirection = bhLight.getDirectionAt(worldPoint);
    attenuatedIntensity = bhLight.getAttenuatedIntensity(worldPoint);
} else if (light instanceof FractalLight) {
    FractalLight fLight = (FractalLight) light;
    lightDirection = fLight.getDirectionAt(worldPoint);
    attenuatedIntensity = fLight.getAttenuatedIntensity(worldPoint);
} else {
    return Color.BLACK;
}

```

// Diffuse component

```

double NdotL = Math.max(0, worldNormal.dot(lightDirection));
int rDiffuse = (int)(baseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int)(baseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int)(baseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

```

// Specular component

```

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectionVec = lightDirection.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectionVec.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

int rSpecular = (int)(specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
int gSpecular = (int)(specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);

```

```

        int bSpecular = (int)(specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

        // Combine components
        int finalR = Math.min(255, rAmbient + rDiffuse + rSpecular);
        int finalG = Math.min(255, gAmbient + gDiffuse + gSpecular);
        int finalB = Math.min(255, bAmbient + bDiffuse + bSpecular);

        return new Color(finalR, finalG, finalB);
    }

    @Override
    public double getReflectivity() {
        return reflectivity;
    }

    @Override
    public double getIndexOfRefraction() {
        return ior;
    }

    @Override
    public double getTransparency() {
        return transparency;
    }

}

/**
Material horizontalRects = new RectangleCheckerMaterial(
Color.WHITE, Color.BLACK,
2.0, 1.0, // Width 2.0, height 1.0
plane.getInverseTransform()
);

Material verticalRects = new RectangleCheckerMaterial(
Color.RED, Color.BLUE,
1.0, 3.0, // Width 1.0, height 3.0
plane.getInverseTransform()

```

```
);

Material customChecker = new RectangleCheckerMaterial(
    new Color(200, 150, 50), // Beige
    new Color(50, 100, 50), // Dark green
    1.5, 0.8,           // Dimensions
    0.15, 0.8, 0.3,    // Ambient, Diffuse, Specular
    25.0, 0.1,          // Shininess, Reflectivity
    1.3, 0.02,          // IOR, Transparency
    plane.getInverseTransform()
);
/*
*/
```

```
// =====
// File: /net/elenamurat/material/StripedMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

//custom imports
import net.elena.murat.light.*;
import net.elena.murat.math.Matrix3;
import net.elena.murat.math.Matrix4;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Ray;

/**
 * A material that creates a repeating striped pattern on a surface.
 * It applies a Phong-like lighting model (Ambient + Diffuse + Specular)
 * to the pattern.
 * This material now fully implements the extended Material interface.
 */
public class StripedMaterial implements Material {

    private final Color color1; // Color for the first stripe
```

```

private final Color color2; // Color for the second stripe
private final double stripeSize; // Controls the width/frequency of the
stripes. Smaller value = thinner/more stripes.
private final StripeDirection direction; // The direction of the stripes (e.g.,
Horizontal, Vertical, Diagonal)
private Matrix4 objectInverseTransform; // The inverse transformation
matrix of the object

// Lighting coefficients (similar to Phong, as in other texture materials)
private final double ambientCoefficient;
private final double diffuseCoefficient;
private final double specularCoefficient;
private final double shininess;
private final double reflectivity;
private final double ior; // Index of Refraction
private final double transparency;

private final Color specularColor = Color.WHITE; // Default specular
color for highlights

/***
 * Full constructor for StripedMaterial.
 * @param color1 First color of the stripes.
 * @param color2 Second color of the stripes.
 * @param stripeSize Controls the width/frequency of the stripes. Smaller
value means more, thinner stripes.
 * @param direction The orientation of the stripes (Horizontal, Vertical,
Diagonal).
 * @param ambientCoefficient Ambient light contribution.
 * @param diffuseCoefficient Diffuse light contribution.
 * @param specularCoefficient Specular light contribution.
 * @param shininess Shininess for specular highlights.
 * @param reflectivity Material's reflectivity (0.0 to 1.0).
 * @param ior Index of refraction for transparent materials.
 * @param transparency Material's transparency (0.0 to 1.0).
 * @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.
 */
public StripedMaterial(Color color1, Color color2, double stripeSize,

```

```
StripeDirection direction,
    double ambientCoefficient, double diffuseCoefficient,
    double specularCoefficient, double shininess,
    double reflectivity, double ior, double transparency,
    Matrix4 objectInverseTransform) { // Added objectInverseTransform
    this.color1 = color1;
    this.color2 = color2;
    this.stripeSize = stripeSize;
    this.direction = direction;
    this.objectInverseTransform = objectInverseTransform; // Store the
    inverse transform
```

```
    this.ambientCoefficient = ambientCoefficient;
    this.diffuseCoefficient = diffuseCoefficient;
    this.specularCoefficient = specularCoefficient;
    this.shininess = shininess;
    this.reflectivity = reflectivity;
    this.ior = ior;
    this.transparency = transparency;
```

```
}
```

```
/**
```

```
* Simplified constructor with default Phong-like coefficients.
* Default direction is VERTICAL.
* @param color1 First color of the stripes.
* @param color2 Second color of the stripes.
* @param stripeSize Controls the width/frequency of the stripes.
* @param objectInverseTransform The full inverse transformation
matrix of the object.
```

```
*/
```

```
public StripedMaterial(Color color1, Color color2, double stripeSize,
Matrix4 objectInverseTransform) { // Added objectInverseTransform
    this(color1, color2, stripeSize, StripeDirection.VERTICAL,
    objectInverseTransform); // Default to vertical stripes
}
```

```
/**
```

```
* Simplified constructor with default Phong-like coefficients and
specified direction.
```

```

* @param color1 First color of the stripes.
* @param color2 Second color of the stripes.
* @param stripeSize Controls the width/frequency of the stripes.
* @param direction The orientation of the stripes.
* @param objectInverseTransform The full inverse transformation
matrix of the object.
*/
public StripedMaterial(Color color1, Color color2, double stripeSize,
StripeDirection direction, Matrix4 objectInverseTransform) { // Added
objectInverseTransform
    // Default Phong parameters, consistent with other texture materials
    this(color1, color2, stripeSize, direction,
        0.1, // ambientCoefficient
        0.8, // diffuseCoefficient
        0.1, // specularCoefficient
        10.0, // shininess
        0.0, // reflectivity
        1.0, // indexOfRefraction
        0.0, // transparency
        objectInverseTransform // Pass the inverse transform
    );
}

```

@Override

```

public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

```

```
/**
```

* Calculates the base pattern color (stripe color) at a given UV coordinate pair.

* The input u and v coordinates are already scaled by stripeSize.

*

* @param u U texture coordinate.

* @param v V texture coordinate.

* @return The pattern color (color1 or color2).

*/

```
private Color getPatternColor(double u, double v) {
```

```

double value; // The coordinate value to check for stripes

switch (direction) {
    case HORIZONTAL:
        value = v; // Stripes vary along the V-axis (latitude)
        break;
    case VERTICAL:
        value = u; // Stripes vary along the U-axis (longitude)
        break;
    case DIAGONAL:
        // Diagonal stripes can be based on u+v or u-v
        value = (u + v) / 2.0; // Average for diagonal, keeps scale consistent
        break;
    default:
        value = u; // Fallback to vertical
        break;
}

// Ensure value is within [0, 1) for consistent repetition check
// Use Math.floor to get the integer part, then check parity
// Add a small epsilon to handle floating point inaccuracies at
boundaries.

if (Math.floor(value + Ray.EPSILON) % 2 == 0) {
    return color1;
} else {
    return color2;
}
}

/***
 * Returns the final shaded color at a given 3D point on the surface,
 * applying the striped pattern and a Phong-like lighting model.
 * Uses spherical UV mapping for better generalization across shapes.
 *
 * @param worldPoint The 3D point on the surface in world coordinates.
 * @param worldNormal The surface normal at that point in world
coordinates (should be normalized).
 * @param light The light source.
 * @param viewerPos The position of the viewer/camera.

```

```

* @return The final shaded color.
*/
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // Check if inverse transform is valid before proceeding
    if (objectInverseTransform == null) {
        System.err.println("Error: StripedMaterial's inverse transform is null.
Returning black.");
        return Color.BLACK;
    }

    // 1. Transform the intersection point from world space into the object's
    // local space
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);

    // --- UV Coordinate Calculation (Spherical Mapping for general curved
    // surfaces) ---
    // Convert localPoint to a Vector3 from the origin (0,0,0) to normalize it.
    // This is crucial for spherical mapping which treats the point as a
    // direction from the center.
    Vector3 P_as_vector = new Vector3(localPoint.x, localPoint.y,
    localPoint.z);
    Vector3 P_normalized = P_as_vector.normalize();

    // Spherical mapping (standard latitude/longitude projection)
    // U: longitude, V: latitude
    // U ranges from 0 to 1, V ranges from 0 to 1
    double u = 0.5 + Math.atan2(P_normalized.z, P_normalized.x) / (2 *
    Math.PI);
    double v = 0.5 - Math.asin(P_normalized.y) / Math.PI;

    // Apply stripeSize as a frequency multiplier to the UV coordinates
    u *= this.stripeSize;
    v *= this.stripeSize;

    // Get the base pattern color from the striped procedural pattern.
    Color patternBaseColor = getPatternColor(u, v); // Pass scaled u,v
}

```

```

// --- Lighting Calculation (Ambient + Diffuse + Specular Phong Model)
---

// This part is consistent with your other material classes.

Color lightColor = light.getColor();
double attenuatedIntensity = 0.0; // Initialize attenuated intensity

// Ambient Light calculation
if (light instanceof ElenaMuratAmbientLight) {
    double ambientIntensity = light.getIntensity();
    // Ambient component is scaled by the pattern color
    int rAmbient = (int) (patternBaseColor.getRed() * ambientCoefficient
* ambientIntensity * (lightColor.getRed() / 255.0));
    int gAmbient = (int) (patternBaseColor.getGreen() *
ambientCoefficient * ambientIntensity * (lightColor.getGreen() / 255.0));
    int bAmbient = (int) (patternBaseColor.getBlue() * ambientCoefficient
* ambientIntensity * (lightColor.getBlue() / 255.0));
    return new Color(
        Math.min(255, rAmbient),
        Math.min(255, gAmbient),
        Math.min(255, bAmbient)
    );
} else { // Directional, Point, Pulsating, and Spot lights (non-ambient)
    Vector3 lightDirection = getLightDirection(light, worldPoint); // Use
worldPoint for light direction
    if (lightDirection == null) { // Handle unsupported light types
        return Color.BLACK;
    }

    // Get attenuated intensity based on light type
    if (light instanceof MuratPointLight) {
        attenuatedIntensity = ((MuratPointLight)
light).getAttenuatedIntensity(worldPoint);
    } else if (light instanceof ElenaDirectionalLight) {
        attenuatedIntensity = ((ElenaDirectionalLight) light).getIntensity();
    } else if (light instanceof PulsatingPointLight) {
        attenuatedIntensity = ((PulsatingPointLight)
light).getAttenuatedIntensity(worldPoint);
    }
}

```

```

    } else if (light instanceof SpotLight) {
        attenuatedIntensity = ((SpotLight)
    light).getAttenuatedIntensity(worldPoint);
    } else if (light instanceof BioluminescentLight) {
        attenuatedIntensity = ((BioluminescentLight)
    light).getAttenuatedIntensity(worldPoint);
    } else if (light instanceof BlackHoleLight) {
        attenuatedIntensity = ((BlackHoleLight)
    light).getAttenuatedIntensity(worldPoint);
    } else if (light instanceof FractalLight) {
        attenuatedIntensity = ((FractalLight)
    light).getAttenuatedIntensity(worldPoint);
    } else {
        // Bu else bloğuna düşmemesi gerekiyor, çünkü getLightDirection
        zaten kontrol ediyor.
        System.err.println("Warning: Unknown or unsupported light type for
StripedMaterial shading (intensity): " + light.getClass().getName());
        return Color.BLACK;
    }

    // Diffuse component
    double NdotL = Math.max(0, worldNormal.dot(lightDirection));
    int rDiffuse = (int) (patternBaseColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
    int gDiffuse = (int) (patternBaseColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
    int bDiffuse = (int) (patternBaseColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

    // Specular component
    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectionVector =
    lightDirection.negate().reflect(worldNormal); // Use negate() before
reflect() for correct reflection vector
    double RdotV = Math.max(0, reflectionVector.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess);

    int rSpecular = (int) (specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);

```

```

    int gSpecular = (int) (specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
    int bSpecular = (int) (specularColor.getBlue() * specularCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

    // Sum up all components for this light source
    int finalR = Math.min(255, rDiffuse + rSpecular); // Ambient is
handled separately
    int finalG = Math.min(255, gDiffuse + gSpecular);
    int finalB = Math.min(255, bDiffuse + bSpecular);

    return new Color(finalR, finalG, finalB);
}

}

/***
 * Returns the reflectivity coefficient of the material.
 * @return The reflectivity value (0.0-1.0).
 */
@Override
public double getReflectivity() { return reflectivity; }

/***
 * Returns the index of refraction (IOR) of the material.
 * @return The index of refraction.
 */
@Override
public double getIndexOfRefraction() { return ior; }

/***
 * Returns the transparency coefficient of the material.
 * @return The transparency value (0.0-1.0).
 */
@Override
public double getTransparency() { return transparency; }

// No need for getShininess() as it's not part of the Material interface and
used internally.
// public double getShininess() { return shininess; }

```

```

/***
 * Helper method to clamp a double value between 0.0 and 1.0.
 * @param val The value to clamp.
 * @return The clamped value.
 */
private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}

/***
 * Helper method to get the normalized light direction vector from a light
source to a point.
 * @param light The light source.
 * @param worldPoint The point in world coordinates.
 * @return Normalized light direction vector, or null if light type is
unsupported.
 */
private Vector3 getLightDirection(Light light, Point3 worldPoint) {
    if (light instanceof MuratPointLight) {
        return
((MuratPointLight)light).getPosition().subtract(worldPoint).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return
((ElenaDirectionalLight)light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return
((PulsatingPointLight)light).getPosition().subtract(worldPoint).normalize()
;
    } else if (light instanceof SpotLight) {
        return ((SpotLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof BioluminescentLight) {
        return
((BioluminescentLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof BlackHoleLight) {
        return
((BlackHoleLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight)light).getDirectionAt(worldPoint).normalize();
    }
}

```

```
        } else {
            System.err.println("Warning: Unknown light type in
PlaneFaceElenaTextureMaterial");
            return new Vector3(0, 1, 0); // Varsayılan yön
        }
    }

}

// =====
// File: /net/elenamurat/material/SuperBrightDebugMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;

public class SuperBrightDebugMaterial implements Material {

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPos) {
        return Color.WHITE; // Always WHITE
    }

    @Override
    public double getReflectivity() {
        return 0.0;
    }

    @Override
    public double getIndexOfRefraction() {
        return 1.0;
    }
}
```

```
@Override
public double getTransparency() {
    return 0.0;
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

}

// =====
// File: /net/elenamurat/material/DamaskCeramicMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class DamaskCeramicMaterial implements Material {
    private final Color primaryColor;
    private final Color secondaryColor;
    private final double shininess;
    private final double ambientCoeff;
    private final double specularCoeff;
    private final double reflectivity=0.1;

    private Matrix4 objectInverseTransform;

    public DamaskCeramicMaterial(Color primary, Color secondary,
        double shininess, Matrix4 invTransform) {
        this(primary, secondary, shininess, 0.1, 0.8, invTransform);
    }
}
```

```
public DamaskCeramicMaterial(Color primary, Color secondary,
    double shininess, double ambient,
    double specular, Matrix4 invTransform) {
    this.primaryColor = primary;
    this.secondaryColor = secondary;
    this.shininess = Math.max(1.0, shininess);
    this.ambientCoeff = ambient;
    this.specularCoeff = specular;
    this.objectInverseTransform = invTransform;
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewPos) {
```

// 1. Coordinate transformation

Point3 localPoint =

```
objectInverseTransform.transformPoint(worldPoint);
```

Vector3 normal =

```
objectInverseTransform.transformNormal(worldNormal).normalize();
```

// 2. Procedural damask pattern

```
double pattern = Math.sin(localPoint.x * 5) *
Math.cos(localPoint.y * 7) *
Math.sin(localPoint.z * 3);
```

```
Color baseColor = pattern > 0 ? primaryColor : secondaryColor;
```

// 3. Light calculations

```
Vector3 lightDir = light.getPosition().subtract(worldPoint).normalize();
```

```
Vector3 viewDir = viewPos.subtract(worldPoint).normalize();
```

```
Vector3 reflectDir = lightDir.negate().reflect(normal);
```

// 4. Components

```
double NdotL = Math.max(0, normal.dot(lightDir));
```

```

double RdotV = Math.max(0, reflectDir.dot(viewDir));

// 5. Light effect (supports colored light)
Color lightColor = light.getColor();
double intensity = light.getIntensityAt(worldPoint);

// 6. Color mixing
Color ambient = ColorUtil.multiplyColors(baseColor, lightColor,
ambientCoeff);
Color diffuse = ColorUtil.multiplyColors(baseColor, lightColor, NdotL
* intensity);
Color specular = ColorUtil.multiplyColors(lightColor,
new Color(255, 255, 255),
Math.pow(RdotV, shininess) * specularCoeff * intensity);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.4; }
@Override public double getTransparency() { return 0.0; }

}

/ ***
// Scene setup
Scene scene = new Scene();

// 1. Ceramic sphere
Sphere ceramicSphere = new Sphere(1.0);
ceramicSphere.setTransform(Matrix4.translate(new Vector3(-2, 0, -5)));
ceramicSphere.setMaterial(new DamaskCeramicMaterial(
new Color(240, 240, 240), // White
new Color(70, 70, 70), // Gray
50.0, // Shininess
ceramicSphere.getInverseTransform()
));
scene.addShape(ceramicSphere);

```

```

// 2. Water sphere
WaterRippleMaterial waterMat = new WaterRippleMaterial(
new Color(80, 180, 220), // Water blue
0.5, // Wave speed
Matrix4.translate(new Vector3(2, 0, -5)).inverse()
);
Sphere waterSphere = new Sphere(1.0);
waterSphere.setTransform(Matrix4.translate(new Vector3(2, 0, -5)));
waterSphere.setMaterial(waterMat);
scene.addShape(waterSphere);

// 3. Light source
scene.addLight(new MuratPointLight(
new Point3(0, 5, 0),
new Color(255, 220, 180), // Warm white
2.0
));

```

// In render loop

```

double deltaTime = 0.016; // ~60 FPS
waterMat.update(deltaTime);
*/

```

```

// =====
// File: /net/elenamurat/material/SmartGlassMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

public class SmartGlassMaterial implements Material {
    private final Color glassColor;
    private final double clarity;

```

```

public SmartGlassMaterial() {
    this(new Color(200, 230, 255), 0.95); // Default settings
}

public SmartGlassMaterial(Color color, double clarity) {
    this.glassColor = color != null ? color : new Color(200, 230, 255);
    this.clarity = Math.max(0.1, Math.min(1.0, clarity));
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // 1. Null Guard Clauses
    if (point == null) point = new Point3(0, 0, 0);
    if (normal == null) normal = new Vector3(0, 1, 0);
    if (viewerPos == null) viewerPos = new Point3(0, 0, -1);

    // 2. Default color (if light is null)
    if (light == null || light.getPosition() == null) {
        return new Color(
            glassColor.getRed(),
            glassColor.getGreen(),
            glassColor.getBlue(),
            200 // 78% opaque
        );
    }

    // 3. Normalization guarantee
    Vector3 safeNormal = normal.length() > 0 ? normal.normalize() : new
Vector3(0, 1, 0);
    Vector3 viewDir = viewerPos.subtract(point).normalizeSafe();
    Vector3 lightDir = light.getPosition().subtract(point).normalizeSafe();

    // 4. Glass effects
    double fresnel = Math.pow(1.0 - Math.abs(safeNormal.dot(viewDir)),
5);
    double specular = Math.pow(Math.max(0, safeNormal.dot(lightDir)),
256 * clarity);
}

```

```

// 5. Color calculation (clamped)
return new Color(
    clamp(glassColor.getRed() * (0.7 + 0.3 * specular)),
    clamp(glassColor.getGreen() * (0.8 + 0.2 * specular)),
    clamp(glassColor.getBlue() * (0.9 + 0.1 * specular)),
    (int)(50 + 205 * clarity) // Alpha: 50-255
);
}

private int clamp(double value) {
    return (int) Math.max(0, Math.min(255, value));
}

@Override public double getReflectivity() { return 0.1; }
@Override public double getIndexOfRefraction() { return 1.5; }
@Override public double getTransparency() { return 0.9; }

@Override
public void setObjectTransform(Matrix4 tm) {

}

// =====
// File: /net/elenamurat/material/ContrastMaterial.java
// =====

package net.elena.murat.material;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import java.awt.Color;

public class ContrastMaterial implements Material {

    private Color baseColor;
    private double contrast;
    private boolean useLightColor;
}

```

```
private double transparency = 0.0; //opaque

public ContrastMaterial(Color baseColor, double contrast) {
    this(baseColor, contrast, false);
}

public ContrastMaterial(Color baseColor, double contrast, boolean
useLightColor) {
    this.baseColor = baseColor;
    this.contrast = contrast;
    this.useLightColor = useLightColor;
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Color sourceColor = baseColor;

    // Işık rengini kullanma seçeneği
    if (useLightColor) {
        sourceColor = light.getColor();
    }

    int alfa = sourceColor.getAlpha ();
    double alpha = ((double)(alfa))/255.0;

    setTransparency (1-alpha);

    // Kontrast uygula
    return applyContrast(sourceColor, contrast);
}

private Color applyContrast(Color color, double contrastFactor) {
    float[] rgb = color.getRGBColorComponents(null);

    // Kontrast formülü: (renk - 0.5) * factor + 0.5
    float r = (float) Math.max(0.0, Math.min(1.0, (rgb[0] - 0.5f) *
contrastFactor + 0.5f));
```

```
    float g = (float) Math.max(0.0, Math.min(1.0, (rgb[1] - 0.5f) *  
contrastFactor + 0.5f));  
    float b = (float) Math.max(0.0, Math.min(1.0, (rgb[2] - 0.5f) *  
contrastFactor + 0.5f));  
  
    return new Color(r, g, b);  
}  
  
{@Override  
public double getReflectivity() {  
    return 0.0;  
}  
  
{@Override  
public double getTransparency() {  
    return transparency;  
}  
  
public void setTransparency (double tnw){  
    this.transparency = tnw;  
}  
  
{@Override  
public double getIndexOfRefraction() {  
    return 1.0;  
}  
  
{@Override  
public void setObjectTransform(Matrix4 tm) {  
    // Transform'a ihtiyaç yok  
}  
  
// Getter ve Setter metodları  
public Color getBaseColor() {  
    return baseColor;  
}  
  
public void setBaseColor(Color baseColor) {  
    this.baseColor = baseColor;
```

```
}

public double getContrast() {
    return contrast;
}

public void setContrast(double contrast) {
    this.contrast = Math.max(0.0, contrast);
}

public boolean isUseLightColor() {
    return useLightColor;
}

public void setUseLightColor(boolean useLightColor) {
    this.useLightColor = useLightColor;
}

// Yardımcı metod: Otomatik kontrast için orta gri renk
public static Color getMiddleGray() {
    return new Color(0.5f, 0.5f, 0.5f);
}

// =====
// File: /net/elenamurat/material/NonScaledTransparentPNGMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

/**
```

```
* Material that textures a surface with a transparent PNG image
* using original pixel dimensions without scaling or tiling.
* Assumes planar UV mapping on XY plane.
*/
public class NonScaledTransparentPNGMaterial implements Material {

    private BufferedImage texture;
    private Matrix4 objectInverseTransform = new Matrix4();
    private double transparency = 1.0;

    private final int originalWidth;
    private final int originalHeight;

    private final double billboardWidth;
    private final double billboardHeight;

    private float gammaCorrection = 2.4f;

    private double shadowAlphaThreshold = 0.1;

    /**
     * Constructor.
     * @param texture BufferedImage with alpha channel (PNG)
     * @param billboardWidth Example 6.6
     * @param billboardHeight Example 3.6
     */
    public NonScaledTransparentPNGMaterial(BufferedImage texture,
                                            double billboardWidth, double billboardHeight) {
        if (texture == null) {
            throw new IllegalArgumentException("Texture cannot be null");
        }
        this.texture = texture;
        this.originalWidth = texture.getWidth();
        this.originalHeight = texture.getHeight();

        this.billboardWidth = billboardWidth;
        this.billboardHeight = billboardHeight;
    }
}
```

```

public void setGammaCorrection (float gamma) {
    this.gammaCorrection = gamma;
}

@Override
public void setObjectTransform(Matrix4 inverseTransform) {
    if (inverseTransform != null) {
        this.objectInverseTransform = inverseTransform;
    } else {
        this.objectInverseTransform = new Matrix4();
    }
}

/**
 * Returns the color at the given world point on the surface.
 * Uses planar UV mapping on XY plane.
 * Fully transparent pixels return Color(0,0,0,0).
 * Applies UV offset.
 *
 * @param point World space point on surface
 * @param normal Surface normal (unused here)
 * @param light Light source (unused here)
 * @param viewerPos Viewer position (unused here)
 * @return Color with alpha channel
 */
// Original
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    if (texture == null) {
        setTransparency(1.0);
        return new Color(0, 0, 0, 0);
    }

    Point3 local = objectInverseTransform.transformPoint(point);

    // Directly map local coordinates to [0, 1] without scaling
    double u = (local.x / billboardWidth) + 0.5; // [-0.5, 0.5] -> [0, 1]
    double v = (local.y / billboardHeight) + 0.5; // [-0.5, 0.5] -> [0, 1]
}

```

```

v = 1.0 - v; // Flip V for image coordinates

// Clamp to [0,1] to prevent sampling outside texture
u = Math.max(0.0, Math.min(1.0, u));
v = Math.max(0.0, Math.min(1.0, v));

int px = (int) (u * (originalWidth - 1));
int py = (int) (v * (originalHeight - 1));

int argb = texture.getRGB(px, py);
int alpha = (argb >> 24) & 0xFF;

if (alpha > 5) {
    setTransparency(0.0);
    int red = (argb >> 16) & 0xFF;
    int green = (argb >> 8) & 0xFF;
    int blue = argb & 0xFF;
    Color linearColor = ColorUtil.sRGBToLinear(new Color(red, green,
blue), gammaCorrection);
    return linearColor;
}

setTransparency(1.0);
return new Color(0, 0, 0, 0);
}

public boolean hasShadowAt(Point3 point) {
    if (texture == null) {
        return false;
    }

    Point3 localPoint = objectInverseTransform.transformPoint(point);

    double nx = localPoint.x / (billboardWidth / 2.0);
    double ny = localPoint.y / (billboardHeight / 2.0);
    if (nx * nx + ny * ny > 1.0) {
        return false;
    }
}

```

```
double u = (localPoint.x + 1.0) * 0.5;
double v = (1.0 - (localPoint.y + 1.0) * 0.5);

u = clamp(u, 0.0, 1.0);
v = clamp(v, 0.0, 1.0);

int xPixel = (int)(u * originalWidth);
int yPixel = (int)(v * originalHeight);

xPixel = Math.min(Math.max(xPixel, 0), originalWidth - 1);
yPixel = Math.min(Math.max(yPixel, 0), originalHeight - 1);

int argb = texture.getRGB(xPixel, yPixel);
int alpha = (argb >> 24) & 0xFF;

return (alpha / 255.0) >= shadowAlphaThreshold;
}

private double clamp(double value, double min, double max) {
    return Math.max(min, Math.min(max, value));
}

@Override
public double getReflectivity() {
    return 0.0;
}

@Override
public double getIndexOfRefraction() {
    return 1.0;
}

@Override
public double getTransparency() {
    return transparency;
}

private void setTransparency(double t) {
    this.transparency = t;
```

```
}

public double getShadowAlphaThreshold() {
    return shadowAlphaThreshold;
}

public void setShadowAlphaThreshold(double threshold) {
    this.shadowAlphaThreshold = clamp(threshold, 0.0, 1.0);
}

public int getOriginalWidth() {
    return originalWidth;
}

public int getOriginalHeight() {
    return originalHeight;
}

public double getBillboardWidth() {
    return billboardWidth;
}

public double getBillboardHeight() {
    return billboardHeight;
}

}

// =====
// File: /net/elenamurat/material/TelmarkPatternMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
```

```
import net.elena.murat.util.ColorUtil;

public class TelemarkPatternMaterial implements Material {
    private final Color baseColor;
    private final Color patternColor;
    private final Color accentColor;
    private final double patternScale;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.45;
    private final double diffuseCoeff = 0.8;
    private final double specularCoeff = 0.12;
    private final double shininess = 18.0;
    private final double reflectivity = 0.07;
    private final double ior = 1.6;
    private final double transparency = 0.0;

    public TelemarkPatternMaterial() {
        this(new Color(0x8B, 0x00, 0x00), new Color(0xFF, 0xD7, 0x00), new
Color(0x00, 0x64, 0x00), 5.0);
    }

    public TelemarkPatternMaterial(Color baseColor, Color patternColor,
Color accentColor, double patternScale) {
        this.baseColor = baseColor;
        this.patternColor = patternColor;
        this.accentColor = accentColor;
        this.patternScale = patternScale;
        this.objectTransform = Matrix4.identity();
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectTransform = tm;
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
```

```

light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateTelemarkPattern(objectPoint);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(Color.WHITE, props.color,
specularCoeff * specFactor);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateTelemarkPattern(Point3 point) {
    double x = point.x * patternScale;
    double y = point.y * patternScale;
    double z = point.z * patternScale;

    // Traditional Telemark geometric patterns
    double diamond1 = Math.abs(Math.sin(x * 2.0) + Math.cos(y * 2.0));
    double diamond2 = Math.abs(Math.sin(x * 3.0 + y * 1.5) + Math.cos(y

```

```

* 2.0 + x * 1.2));
double cross = (Math.floor(x * 1.5) + Math.floor(y * 1.5)) % 3.0;
double border = Math.abs(Math.sin(x * 4.0) * Math.cos(y * 4.0));

double combinedPattern = (diamond1 * 0.3 + diamond2 * 0.25 + cross *
0.25 + border * 0.2);
double normalizedPattern = combinedPattern % 1.0;

if (normalizedPattern < 0.3) {
    // Base color with diamond pattern
    return baseColor;
} else if (normalizedPattern < 0.6) {
    // Main geometric pattern
    return patternColor;
} else if (normalizedPattern < 0.8) {
    // Accent details
    return accentColor;
} else {
    // Border and outline elements
    return ColorUtil.darkenColor(patternColor, 0.4);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

```

```
// =====
// File: /net/elenamurat/material/DiamondMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.util.Random;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

public class DiamondMaterial implements Material {
    private final Color baseColor;
    private final double indexOfRefraction;
    private final double baseReflectivity;
    private final double baseTransparency;
    private final Random random;

    private double currentReflectivity;
    private double currentTransparency;
    private final double dispersionStrength;
    private final double fireEffect;

    public DiamondMaterial(Color baseColor, double ior,
        double reflectivity, double transparency,
        double dispersionStrength, double fireEffect) {
        this.baseColor = baseColor;
        this.indexOfRefraction = ior;
        this.baseReflectivity = reflectivity;
        this.baseTransparency = transparency;
        this.dispersionStrength = dispersionStrength;
        this.fireEffect = fireEffect;
        this.random = new Random();
    }
}
```

```

public DiamondMaterial(Color baseColor, double ior) {
    this(baseColor, ior, 0.15, 0.98, 0.3, 0.7);
}

public DiamondMaterial(double ior) {
    this(new Color(255, 250, 245), ior);
}

public DiamondMaterial() {
    this(2.42); // Diamond IOR
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    Vector3 viewDir = viewerPos.subtract(point).normalize();
    Vector3 lightDir = light.getDirectionTo(point).normalize();

    double fresnel = calculateEnhancedFresnel(viewDir, normal, 1.0,
indexOfRefraction);

    this.currentReflectivity = Math.min(0.97, baseReflectivity + (fresnel *
0.85));
    this.currentTransparency = Math.max(0.02, baseTransparency * (1.0 -
fresnel * 0.1));

    double NdotL = Math.max(0.4, normal.dot(lightDir));
    double intensity = light.getIntensityAt(point);

    Vector3 reflectDir = lightDir.reflect(normal);
    double specular = Math.pow(Math.max(0.0, viewDir.dot(reflectDir)),
256);

    Color diamondTint = ColorUtil.multiplyColor(baseColor, 0.9);
    Color diffuse = ColorUtil.multiplyColor(diamondTint, NdotL * 0.3 *
intensity);

    Color specularHighlight = ColorUtil.multiplyColor(light.getColor(),
specular * 2.0 * intensity);
}

```

```
    Color dispersionEffect = applyDispersionEffect(specularHighlight,  
fresnel);
```

```
    Color result = ColorUtil.addSafe(diffuse, specularHighlight);  
    result = ColorUtil.addSafe(result, dispersionEffect);
```

```
    return ColorUtil.clampColor(result);
```

```
}
```

```
private double calculateEnhancedFresnel(Vector3 viewDir, Vector3  
normal,
```

```
    double ior1, double ior2) {
```

```
    double cosTheta = Math.abs(viewDir.dot(normal));
```

```
    cosTheta = Math.max(0.0, Math.min(1.0, cosTheta));
```

```
    double r0 = Math.pow((ior1 - ior2) / (ior1 + ior2), 2);
```

```
    double fresnel = r0 + (1.0 - r0) * Math.pow(1.0 - cosTheta, 3.5);
```

```
    return Math.max(0.0, Math.min(1.0, fresnel));
```

```
}
```

```
private Color applyDispersionEffect(Color baseColor, double fresnel) {  
    if (dispersionStrength <= 0) return new Color(0, 0, 0, 0);
```

```
    double strength = dispersionStrength * fresnel * fireEffect;
```

```
    int r = (int) (strength * 180 * (0.8 + random.nextDouble() * 0.4));
```

```
    int g = (int) (strength * 120 * (0.7 + random.nextDouble() * 0.6));
```

```
    int b = (int) (strength * 200 * (0.9 + random.nextDouble() * 0.2));
```

```
    return new Color(
```

```
        Math.min(255, r),
```

```
        Math.min(255, g),
```

```
        Math.min(255, b),
```

```
        (int) (strength * 200)
```

```
    );
```

```
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
}

@Override
public double getReflectivity() {
    return currentReflectivity;
}

@Override
public double getTransparency() {
    return currentTransparency;
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

public Color getColorForRefraction() {
    int r = Math.min(255, (int)(baseColor.getRed() * 0.8 + fireEffect * 20));
    int g = Math.min(255, (int)(baseColor.getGreen() * 0.8 + fireEffect *
10));
    int b = Math.min(255, (int)(baseColor.getBlue() * 0.8 + fireEffect *
30));

    return new Color(r, g, b);
}

public Color getDiamondColor() {
    return baseColor;
}

public double getDispersionStrength() {
    return dispersionStrength;
}

public double getFireEffect() {
    return fireEffect;
}
```

```
}

}

// =====
// File: /net/elenamurat/material/TransparentPNGMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.Light;
import net.elena.murat.math.Matrix4;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

/***
 * Material class that textures a surface with a transparent PNG image.
 * Supports alpha channel and returns fully transparent color for
transparent pixels.
 * Uses planar UV mapping on XY plane (Z ignored).
 *
 * Assumes UV coordinates are derived from object local coordinates
mapped from [-1,1] to [0,1].
 * The objectInverseTransform is used to convert world coordinates to
local object space.
 *
 * Supports UV offset, scale, and optional repeating of the texture.
 * Includes strict alpha handling for complete transparency.
 */
public class TransparentPNGMaterial implements Material {

    private BufferedImage texture;
    private Matrix4 objectInverseTransform = new Matrix4(); // Identity by
default
```

```
private double transparency = 1.0;

// UV offset and scale parameters with default values (no offset, scale=1)
private double uOffset = 0.0;
private double vOffset = 0.0;
private double uScale = 1.0;
private double vScale = 1.0;

// Flag to enable repeating the texture outside [0,1] UV range
private boolean isRepeatTexture = false;

/***
 * Constructor with texture image.
 * @param texture BufferedImage with alpha channel (PNG)
 */
public TransparentPNGMaterial(BufferedImage texture) {
    this.texture = texture;
}

/***
 * Constructor with texture image and UV offset/scale parameters.
 * @param texture BufferedImage with alpha channel (PNG)
 * @param uOffset Horizontal offset for texture coordinates (0.0 = no
offset)
 * @param vOffset Vertical offset for texture coordinates (0.0 = no
offset)
 * @param uScale Horizontal scale factor (1.0 = original size)
 * @param vScale Vertical scale factor (1.0 = original size)
 * @param isRepeatTexture Whether to repeat the texture outside [0,1]
UV range
 */
public TransparentPNGMaterial(BufferedImage texture, double uOffset,
double vOffset, double uScale, double vScale, boolean isRepeatTexture) {
    this.texture = texture;
    this.uOffset = uOffset;
    this.vOffset = vOffset;
    this.uScale = (uScale > 0.0) ? uScale : 1.0; // Prevent zero or negative
scale
    this.vScale = (vScale > 0.0) ? vScale : 1.0;
}
```

```

    this.isRepeatTexture = isRepeatTexture;
}

/***
 * Default constructor (no texture)
 */
public TransparentPNGMaterial() {
    this.texture = null;
}

/***
 * Sets the inverse transform matrix of the object.
 * Used to convert world coordinates to local object space.
 * @param inverseTransform Matrix4 inverse transform
 */
@Override
public void setObjectTransform(Matrix4 inverseTransform) {
    if (inverseTransform != null) {
        this.objectInverseTransform = inverseTransform;
    } else {
        this.objectInverseTransform = new Matrix4(); // Identity fallback
    }
}

/***
 * Returns the color at the given world point on the surface.
 * Uses planar UV mapping on XY plane.
 * Fully transparent pixels return Color(0,0,0,0).
 * Applies UV offset, scale, and optional repeating.
 * Uses strict alpha checking: any alpha less than 255 returns full
transparency.
 *
 * @param point World space point on surface
 * @param normal Surface normal (unused here)
 * @param light Light source (unused here)
 * @param viewerPos Viewer position (unused here)
 * @return Color with alpha channel
 */
@Override

```

```

public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    if(texture == null) {
        setTransparency(1.0); // Fully transparent
        return new Color(0, 0, 0, 0);
    }

    // Transform world coordinates to local object space
    Point3 local = objectInverseTransform.transformPoint(point);

    // Base UV coordinates mapped from [-1,1] to [0,1]
    double u = (local.x + 1.0) * 0.5;
    double v = (1.0 - (local.y + 1.0)) * 0.5;

    // Apply UV scale and offset
    double scaledU = u / uScale + uOffset;
    double scaledV = v / vScale + vOffset;

    // DEBUG
    //if (Math.random() < 0.0001) System.out.println("Local: " + local +
    //      ", U: " + u + ", V: " + v +
    //      ", ScaledU: " + scaledU + ", ScaledV: " + scaledV);

    double finalU, finalV;

    if(isRepeatTexture) {
        // Wrap UVs for tiling (repeat)
        finalU = scaledU - Math.floor(scaledU);
        finalV = scaledV - Math.floor(scaledV);
    } else {
        // No tiling: if UV outside [0,1], return fully transparent color
        if(scaledU < 0.0 || scaledU > 1.0 || scaledV < 0.0 || scaledV > 1.0) {
            setTransparency(1.0); // Fully transparent
            return new Color(0, 0, 0, 0);
            //setTransparency(0.0);
            //return new Color(1f, 0f, 0f, 1f);
        }
        finalU = scaledU;
        finalV = scaledV;
    }
}

```

```

}

//double margin = 0.05;
//finalU = margin + finalU * (1.0 - 2 * margin);
//finalV = margin + finalV * (1.0 - 2 * margin);

// Calculate texture coordinates
int px = (int) (finalU * (texture.getWidth() - 1));
int py = (int) (finalV * (texture.getHeight() - 1));

// Ensure coordinates are within texture bounds
px = Math.max(0, Math.min(texture.getWidth() - 1, px));
py = Math.max(0, Math.min(texture.getHeight() - 1, py));

// Get pixel color with alpha channel
int argb = texture.getRGB(px, py);
int alpha = (argb >> 24) & 0xFF;

// Otherwise return fully transparent color
if (alpha > 5) {
    int red = (argb >> 16) & 0xFF;
    int green = (argb >> 8) & 0xFF;
    int blue = argb & 0xFF;
    setTransparency(0.0); // Fully opaque
    return new Color(red, green, blue, 255);
}

// For any alpha value less than 255, return fully transparent
setTransparency(1.0); // Fully transparent
return new Color(0, 0, 0, 0);
}

/***
 * Returns reflectivity of the material.
 * @return 0.0 (non-reflective)
 */
@Override
public double getReflectivity() {
    return 0.0;
}

```

```
}

/***
 * Returns index of refraction.
 * @return 1.0 (no refraction)
 */
@Override
public double getIndexOfRefraction() {
    return 1.0;
}

/***
 * Returns transparency of the material.
 * @return transparency value (0.0 = opaque, 1.0 = fully transparent)
 */
@Override
public double getTransparency() {
    return transparency;
}

/***
 * Sets the transparency value.
 * @param transparency transparency value (0.0 = opaque, 1.0 = fully
transparent)
 */
private void setTransparency(double transparency) {
    this.transparency = transparency;
}

/***
 * Gets the horizontal texture offset.
 * @return U offset value
 */
public double getUOffset() {
    return uOffset;
}

/***
 * Gets the vertical texture offset.

```

```
* @return V offset value
*/
public double getVOffset() {
    return vOffset;
}

/***
 * Gets the horizontal texture scale factor.
 * @return U scale factor
*/
public double getUScale() {
    return uScale;
}

/***
 * Gets the vertical texture scale factor.
 * @return V scale factor
*/
public double getVScale() {
    return vScale;
}

/***
 * Checks if texture repeating is enabled.
 * @return true if texture repeating is enabled, false otherwise
*/
public boolean isRepeatTexture() {
    return isRepeatTexture;
}

/***
 * Sets whether texture repeating is enabled.
 * @param repeat true to enable repeating, false to disable
*/
public void setRepeatTexture(boolean repeat) {
    this.isRepeatTexture = repeat;
}

/***
```

```
* Sets the texture image.  
* @param texture BufferedImage with alpha channel  
*/  
public void setTexture(BufferedImage texture) {  
    this.texture = texture;  
}  
  
/**  
 * Gets the current texture image.  
 * @return current texture image  
*/  
public BufferedImage getTexture() {  
    return texture;  
}  
}
```

```
// ======  
// File: /net/elenamurat/material/KilimRosemalingMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.*;  
import net.elena.murat.util.ColorUtil;  
  
public class KilimRosemalingMaterial implements Material {  
    private final Color kilimColor;  
    private final Color rosemalingColor;  
    private final Color accentColor;  
    private final double patternIntensity;  
    private Matrix4 objectTransform;  
  
    private final double ambientCoeff = 0.5;  
    private final double diffuseCoeff = 0.85;
```

```
private final double specularCoeff = 0.1;
private final double shininess = 15.0;
private final double reflectivity = 0.06;
private final double ior = 1.5;
private final double transparency = 0.0;

public KilimRosemalingMaterial() {
    this(new Color(0xC4, 0x00, 0x00), new Color(0x00, 0x64, 0x64), new
Color(0xFF, 0xD7, 0x00), 0.7);
}
```

```
public KilimRosemalingMaterial(Color kilimColor, Color
rosemalingColor, Color accentColor, double patternIntensity) {
    this.kilimColor = kilimColor;
    this.rosemalingColor = rosemalingColor;
    this.accentColor = accentColor;
    this.patternIntensity = Math.max(0, Math.min(1, patternIntensity));
    this.objectTransform = Matrix4.identity();
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);
```

```
    Color surfaceColor = calculateFusionPattern(objectPoint);
```

```
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
```

```
    if (props == null) return surfaceColor;
```

```
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
```

```

ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(Color.WHITE, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateFusionPattern(Point3 point) {
    double x = point.x * 12.0;
    double y = point.y * 12.0;
    double z = point.z * 12.0;

    // Kilim geometric patterns (Turkish)
    double kilimPattern1 = Math.abs(Math.sin(x * 2.0) + Math.cos(y *
2.0));
    double kilimPattern2 = (Math.floor(x * 1.2) + Math.floor(y * 1.2)) %
2.0;
    double kilimPattern3 = Math.abs(Math.sin(x * 3.0 + y * 2.0));

    // Rosemaling flower patterns (Norwegian)
    double rosePattern1 = Math.sin(x * 1.5) * Math.cos(y * 1.5 +
Math.sin(z * 0.8));
    double rosePattern2 = Math.abs(Math.sin(x * 2.5 + y * 1.8) +
Math.cos(y * 2.2));
    double rosePattern3 = Math.abs(Math.cos(x * 1.8 + y * 2.0 + z * 1.2));
}

```

```

// Combine both cultural patterns
double kilimWeight = 0.5 * patternIntensity;
double roseWeight = 0.5 * patternIntensity;

double combinedPattern = (kilimPattern1 * 0.2 + kilimPattern2 * 0.15 +
kilimPattern3 * 0.15) * kilimWeight +
(rosePattern1 * 0.2 + rosePattern2 * 0.15 + rosePattern3 * 0.15) *
roseWeight;

double normalizedPattern = (combinedPattern + 1.0) * 0.5;

if (normalizedPattern < 0.3) {
    // Kilim base background
    return kilimColor;
} else if (normalizedPattern < 0.6) {
    // Rosemaling flower elements
    double intensity = (normalizedPattern - 0.3) / 0.3;
    return ColorUtil.blendColors(rosemalingColor,
ColorUtil.lightenColor(rosemalingColor, 0.2), intensity);
} else if (normalizedPattern < 0.8) {
    // Accent details (shared cultural elements)
    return accentColor;
} else {
    // Border and outline elements (fusion pattern)
    double intensity = (normalizedPattern - 0.8) / 0.2;
    Color borderColor = ColorUtil.blendColors(kilimColor,
rosemalingColor, 0.5);
    return ColorUtil.darkenColor(borderColor, intensity * 0.4);
}
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

```

```
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/lena/murat/material/WoodMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class WoodMaterial implements Material {
    private final Color baseColor;
    private final Color grainColor;
    private final double grainFrequency;
    private final double ringVariation;
    private Matrix4 objectInverseTransform;

    // Phong parameters
    private final double ambientCoeff;
    private final double diffuseCoeff;
    private final double specularCoeff;
    private final double shininess;
    private final double reflectivity;
    private final double ior;
    private final double transparency;
    private final Color specularColor = new Color(200, 200, 180);
```

```
public WoodMaterial(Color baseColor, Color grainColor, double  
grainFrequency,  
    double ringVariation, Matrix4 objectInverseTransform) {  
    this(baseColor, grainColor, grainFrequency, ringVariation,  
        0.15, 0.7, 0.15, 15.0, 0.05, 1.3, 0.0, objectInverseTransform);  
}  
  
public WoodMaterial(Color baseColor, Color grainColor, double  
grainFrequency,  
    double ringVariation, double ambientCoeff, double diffuseCoeff,  
    double specularCoeff, double shininess, double reflectivity,  
    double ior, double transparency, Matrix4 objectInverseTransform) {  
    this.baseColor = baseColor;  
    this.grainColor = grainColor;  
    this.grainFrequency = grainFrequency;  
    this.ringVariation = Math.max(0, Math.min(1, ringVariation));  
    this.objectInverseTransform = objectInverseTransform;  
    this.ambientCoeff = ambientCoeff;  
    this.diffuseCoeff = diffuseCoeff;  
    this.specularCoeff = specularCoeff;  
    this.shininess = shininess;  
    this.reflectivity = reflectivity;  
    this.ior = ior;  
    this.transparency = transparency;  
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4();  
    this.objectInverseTransform = tm;  
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light  
light, Point3 viewerPos) {  
    // 1. Get wood texture color  
    Point3 localPoint =  
    objectInverseTransform.transformPoint(worldPoint);  
    Color woodColor = calculateWoodColor(localPoint);
```

```

// 2. Unified light handling
LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
if (props == null) return woodColor;

// 3. Phong shading components
Vector3 lightDir = props.direction;
Color lightColor = props.color;
double intensity = props.intensity;

// Ambient
Color ambient = ColorUtil.multiplyColors(woodColor, lightColor,
ambientCoeff);

// Diffuse
double NdotL = Math.max(0, worldNormal.dot(lightDir));
Color diffuse = ColorUtil.multiplyColors(woodColor, lightColor,
diffuseCoeff * NdotL * intensity);

// Specular
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * intensity;
Color specular = ColorUtil.multiplyColors(specularColor, lightColor,
specularCoeff * specFactor);

// Combine components
return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateWoodColor(Point3 localPoint) {
    double x = localPoint.x, y = localPoint.y, z = localPoint.z;
    double distance = Math.sqrt(x*x + z*z) * grainFrequency;
    double noise = Math.sin(y * 2.0) * ringVariation;
    double ringPattern = Math.pow(Math.sin(distance + noise) * 0.5 + 0.5,
3.0);
    return ColorUtil.blendColors(baseColor, grainColor, ringPattern);
}

```

```

}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }
}

/***
Material wood = new WoodMaterial(
new Color(139, 69, 19), // baseColor: Brown (classic wood color)
new Color(101, 67, 33), // grainColor: Darker vein color
0.5, // grainFrequency: Vein density (0.5 medium density)
0.3, // ringVariation: Annual ring variation
Matrix4.scale(0.1, 0.1, 0.1) // objectInverseTransform: Scale down wood
texture
.multiply(Matrix4.rotateY(Math.toRadians(45))) // Rotate texture by 45°
);

// Alternative lighter colored oak wood:
Material oakWood = new WoodMaterial(
new Color(210, 180, 140), // Light beige
new Color(160, 130, 90), // Medium brown veins
0.7, // Denser veins
0.2, // Less ring variation
Matrix4.identity() // Use texture as-is
);
*/

```

```

// =====
// File: /net/lena/murat/material/HologramDataMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.Light;

```

```

import net.elena.murat.math.*;

public class HologramDataMaterial implements Material {
    private final double dataDensity;
    private final int resolution;
    private final long startTime;

    public HologramDataMaterial(double dataDensity, int resolution) {
        this.dataDensity = Math.max(0.1, Math.min(1.0, dataDensity));
        this.resolution = Math.max(64, Math.min(512, resolution));
        this.startTime = System.currentTimeMillis();
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPos) {
        // 1. Grid position
        int gridX = (int)(point.x * resolution) % resolution;
        int gridY = (int)(point.z * resolution) % resolution;

        // 2. Time-based animation
        double time = (System.currentTimeMillis() - startTime) * 0.001;
        int animOffset = (int)(time * 10) % 10;

        // 3. Data pattern (ASCII art like)
        boolean isActive = (gridX + gridY + animOffset) % 4 == 0 &&
        Math.random() < dataDensity;

        // 4. Glitch effect
        double glitch = Math.sin(time * 3 + point.y * 10) * 0.1;

        return isActive ?
            new Color(
                (int)(100 + 155 * Math.abs(Math.sin(time + point.x))),
                (int)(200 + 55 * Math.abs(Math.cos(time + point.z))),
                255,
                180
            ) :
            new Color(0, 10, 20, 50); // Background color
    }
}

```

```

}

@Override public double getReflectivity() { return 0.3; }
@Override public double getIndexOfRefraction() { return 1.1; }
@Override public double getTransparency() { return 0.8; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

/***
EMShape dataCube = new Cube()
.setMaterial(new HologramDataMaterial(0.7, 256));
*/
// =====
// File: /net/elenamurat/material/EdgeLightColorMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

public class EdgeLightColorMaterial implements Material {

    private float edgeThreshold = 0.2f;
    private Color edgeColor = new Color(30, 30, 30);
    private Color baseColor = new Color(200, 200, 200);

    private static final Vector3[] KERNEL_OFFSETS = {
        new Vector3(-1, -1, 0), new Vector3(0, -1, 0), new Vector3(1, -1, 0),
        new Vector3(-1, 0, 0), new Vector3(0, 0, 0), new Vector3(1, 0, 0),
        new Vector3(-1, 1, 0), new Vector3(0, 1, 0), new Vector3(1, 1, 0)
    }
}

```

```

};

private static final float[] KERNEL_WEIGHTS = {
    0.0f, -1.0f, 0.0f,
    -1.0f, 4.0f, -1.0f,
    0.0f, -1.0f, 0.0f
};

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    // Kernel-based edge detection
    float edgeStrength = calculateKernelEdgeStrength(point, normal,
viewerPoint);

    if (edgeStrength > edgeThreshold) {
        return edgeColor;
    } else {
        // Normal lighting calculation
        Vector3 lightDirection = light.getDirectionTo(point).normalize();
        double intensity = light.getIntensityAt(point);
        float dotProduct = (float) Math.max(0, normal.dot(lightDirection));

        // Calculate final color with lighting
        Color lightColor = light.getColor();
        float lightFactor = (float) (intensity * dotProduct);

        int r = (int) (baseColor.getRed() * lightFactor * lightColor.getRed() /
255);
        int g = (int) (baseColor.getGreen() * lightFactor *
lightColor.getGreen() / 255);
        int b = (int) (baseColor.getBlue() * lightFactor * lightColor.getBlue() /
255);

        return new Color(
            Math.min(255, Math.max(0, r)),
            Math.min(255, Math.max(0, g)),
            Math.min(255, Math.max(0, b))
        );
    }
}

```

```

        }

    /**
     * Kernel-based edge detection using normal and view direction
     * variations
     * Simulates the effect of a Laplacian kernel on the surface
     */
    private float calculateKernelEdgeStrength(Point3 point, Vector3
centerNormal, Point3 viewerPoint) {
        float totalEdgeValue = 0.0f;

        for (int i = 0; i < KERNEL_OFFSETS.length; i++) {
            // Create sample point using kernel offset
            Vector3 offset = KERNEL_OFFSETS[i].multiply(0.05); // Small offset
            for sampling
            Point3 samplePoint = point.add(offset);

            // Calculate normal variation at sample point
            // This simulates what a normal buffer would give us
            Vector3 sampleNormal = calculateSampleNormal(centerNormal,
            offset);

            // Calculate depth variation (simulated)
            float depthVariation = calculateDepthVariation(point, samplePoint,
            viewerPoint);

            // Combine normal and depth variations with kernel weight
            float variation = (float)
            (centerNormal.subtract(sampleNormal).length() * 0.7 + depthVariation *
            0.3);
            totalEdgeValue += variation * KERNEL_WEIGHTS[i];
        }

        return Math.abs(totalEdgeValue);
    }

    /**
     * Simulates normal variation based on offset direction

```

```

*/
private Vector3 calculateSampleNormal(Vector3 centerNormal, Vector3 offset) {
    // Add some noise/variation to the normal based on offset
    double variation = offset.length() * 0.3;
    return new Vector3(
        centerNormal.x + offset.x * variation,
        centerNormal.y + offset.y * variation,
        centerNormal.z + offset.z * variation
    ).normalize();
}

/**
 * Simulates depth variation for edge detection
 */
private float calculateDepthVariation(Point3 center, Point3 sample,
Point3 viewer) {
    // Calculate depth differences (simulated)
    double centerDepth = center.subtract(viewer).length();
    double sampleDepth = sample.subtract(viewer).length();
    return (float) Math.abs(centerDepth - sampleDepth) * 2.0f;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    // Transformation handling if needed
}

@Override
public double getTransparency() {
    return 0.1;
}

@Override
public double getReflectivity() {
    return 0.15;
}

@Override

```

```
public double getIndexOfRefraction() {
    return 1.0;
}

public void setEdgeThreshold(float threshold) {
    this.edgeThreshold = Math.max(0.0f, Math.min(1.0f, threshold));
}

public void setEdgeColor(Color color) {
    this.edgeColor = color;
}

public void setBaseColor(Color color) {
    this.baseColor = color;
}

}
```

```
// =====
// File: /net/elenamurat/material/HotCopperMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class HotCopperMaterial implements Material {
    private final Color copperColor;
    private final Color patinaColor;
    private final double patinaAmount;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.35;
    private final double diffuseCoeff = 0.65;
```

```

private final double specularCoeff = 0.4;
private final double shininess = 45.0;
private final double reflectivity = 0.25;
private final double ior = 2.6;
private final double transparency = 0.0;

public HotCopperMaterial() {
    this(new Color(0xB8, 0x73, 0x33), new Color(0x33, 0x99, 0x77), 0.25);
}

public HotCopperMaterial(Color copperColor, Color patinaColor, double
patinaAmount) {
    this.copperColor = copperColor;
    this.patinaColor = patinaColor;
    this.patinaAmount = Math.max(0, Math.min(1, patinaAmount));
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
    objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculatePatinaPattern(objectPoint);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

```

```

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(new Color(0xFF, 0xE6,
0xC9), props.color, specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculatePatinaPattern(Point3 point) {
    double noise1 = simplexNoise3D(point.x * 8, point.y * 8, point.z * 8);
    double noise2 = simplexNoise3D(point.x * 3 + 5.3, point.y * 3 + 2.7,
point.z * 3 + 1.9);

    double pattern = (noise1 * 0.7 + noise2 * 0.3 + 1) * 0.5;

    if (pattern < patinaAmount) {
        double intensity = pattern / patinaAmount;
        return interpolateColor(patinaColor,
            new Color(Math.min(255, (int)(patinaColor.getRed() * 1.2)),
Math.min(255, (int)(patinaColor.getGreen() * 0.9)),
Math.min(255, (int)(patinaColor.getBlue() * 1.1))),
intensity);
    } else {
        double intensity = (pattern - patinaAmount) / (1 - patinaAmount);
        return interpolateColor(copperColor,
            new Color(Math.min(255, (int)(copperColor.getRed() * 1.1)),
Math.min(255, (int)(copperColor.getGreen() * 0.95)),
Math.min(255, (int)(copperColor.getBlue() * 0.9))),
intensity);
    }
}

```

```
}
```

```
private double simplexNoise3D(double x, double y, double z) {  
    double value = Math.sin(x * 0.472) + Math.cos(y * 0.683) + Math.sin(z  
    * 0.291);  
    value += Math.cos(x * 1.732 + y * 0.846) * 0.6;  
    value += Math.sin(y * 1.357 + z * 2.173) * 0.3;  
    return value % 1.0;  
}
```

```
private Color interpolateColor(Color c1, Color c2, double t) {  
    t = Math.max(0, Math.min(1, t));  
    int r = (int)(c1.getRed() * (1-t) + c2.getRed() * t);  
    int g = (int)(c1.getGreen() * (1-t) + c2.getGreen() * t);  
    int b = (int)(c1.getBlue() * (1-t) + c2.getBlue() * t);  
    return new Color(r, g, b);  
}
```

```
@Override  
public double getReflectivity() {  
    return reflectivity;  
}
```

```
@Override  
public double getIndexOfRefraction() {  
    return ior;  
}
```

```
@Override  
public double getTransparency() {  
    return transparency;  
}
```

```
}
```

```
// ======  
// File: /net/elenamurat/material/ReflectiveMaterial.java
```

```
// ======

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

/***
 * Reflective metallic material class
 */
public class ReflectiveMaterial implements Material {

    private final Color baseColor;
    private final double reflectivity;
    private final double roughness;

    /***
     * Default reflective material (silver color, 70% reflectivity)
     */
    public ReflectiveMaterial() {
        this(new Color(200, 200, 200), 0.7, 0.1);
    }

    /***
     * Customizable reflective material
     * @param baseColor Base color
     * @param reflectivity Reflectivity ratio (0-1)
     * @param roughness Surface roughness (0-1, 0=perfect mirror)
     */
    public ReflectiveMaterial(Color baseColor, double reflectivity, double roughness) {
        this.baseColor = baseColor;
        this.reflectivity = Math.max(0, Math.min(1, reflectivity));
        this.roughness = Math.max(0, Math.min(1, roughness));
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
```

```

Point3 viewerPos) {
    // Corrected light direction calculation (using getDirectionTo)
    Vector3 lightDir = light.getDirectionTo(point).normalize();
    double diffuse = Math.max(0, normal.dot(lightDir));

    // Reflection brightness (Blinn-Phong)
    Vector3 viewDir = viewerPos.subtract(point).normalize();
    Vector3 halfDir = lightDir.add(viewDir).normalize();
    double specular = Math.pow(Math.max(0, normal.dot(halfDir)), 32 /
(roughness + 0.01));

    // Color calculation
    int r = (int)(baseColor.getRed() * diffuse + 255 * specular);
    int g = (int)(baseColor.getGreen() * diffuse + 255 * specular);
    int b = (int)(baseColor.getBlue() * diffuse + 255 * specular);

    return new Color(
        Math.min(255, r),
        Math.min(255, g),
        Math.min(255, b)
    );
}

```

```

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```

@Override
public double getIndexOfRefraction() {
    return 1.0; // IOR irrelevant for metals
}

```

```

@Override
public double getTransparency() {
    return 0.0; // Opaque material
}

```

```

@Override

```

```
public void setObjectTransform(Matrix4 tm) {  
}  
  
// Helper methods  
public static ReflectiveMaterial gold() {  
    return new ReflectiveMaterial(new Color(255, 215, 0), 0.85, 0.15);  
}  
  
public static ReflectiveMaterial silver() {  
    return new ReflectiveMaterial(new Color(192, 192, 192), 0.9, 0.1);  
}  
  
public static ReflectiveMaterial copper() {  
    return new ReflectiveMaterial(new Color(184, 115, 51), 0.8, 0.2);  
}  
}
```

```
// =====  
// File: /net/elenamurat/material/TulipFjordMaterial.java  
// =====
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.*;  
import net.elena.murat.util.ColorUtil;  
  
public class TulipFjordMaterial implements Material {  
    private final Color tulipColor;  
    private final Color fjordColor;  
    private final Color stemColor;  
    private final double bloomIntensity;  
    private Matrix4 objectTransform;  
  
    private final double ambientCoeff = 0.4;
```

```
private final double diffuseCoeff = 0.75;
private final double specularCoeff = 0.35;
private final double shininess = 50.0;
private final double reflectivity = 0.2;
private final double ior = 1.55;
private final double transparency = 0.1;

public TulipFjordMaterial() {
    this(new Color(0xFF, 0x00, 0x00), new Color(0x00, 0x7F, 0xFF), new
Color(0x22, 0x8B, 0x22), 0.6);
}
```

```
public TulipFjordMaterial(Color tulipColor, Color fjordColor, Color
stemColor, double bloomIntensity) {
    this.tulipColor = tulipColor;
    this.fjordColor = fjordColor;
    this.stemColor = stemColor;
    this.bloomIntensity = Math.max(0, Math.min(1, bloomIntensity));
    this.objectTransform = Matrix4.identity();
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);
```

```
    Color surfaceColor = calculateTulipFjordPattern(objectPoint,
worldNormal, viewerPos);
```

```
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;
```

```

Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(new Color(0xFF, 0xF5,
0xEE), props.color, specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateTulipFjordPattern(Point3 point, Vector3 normal,
Point3 viewerPos) {
    double x = point.x * 10.0;
    double y = point.y * 10.0;
    double z = point.z * 10.0;

    // Tulip petal patterns (organic curves)
    double petal1 = Math.sin(x * 1.8 + Math.cos(y * 1.2) * 2.5);
    double petal2 = Math.abs(Math.cos(x * 2.0 + y * 1.5) + Math.sin(y * 1.8
+ z * 1.0));
    double petal3 = Math.sin(x * 2.5 + y * 2.0) * Math.cos(y * 1.2 + z *
0.8);

    // Fjord water patterns (flowing waves)
    double fjord1 = Math.sin(x * 1.5 + Math.sin(y * 0.7) * 1.8);
    double fjord2 = Math.cos(y * 1.3 + Math.cos(x * 0.9) * 2.2);
    double fjord3 = Math.abs(Math.sin(x * 2.2 + y * 1.6 + z * 1.3));
}

```

```

// Cultural fusion pattern
double tulipWeight = bloomIntensity;
double fjordWeight = 1.0 - bloomIntensity;

double combinedPattern = (petal1 * 0.25 + petal2 * 0.2 + petal3 * 0.15)
* tulipWeight +
(fjord1 * 0.2 + fjord2 * 0.15 + fjord3 * 0.05) * fjordWeight;

double normalizedPattern = (combinedPattern + 1.0) * 0.5;

// View-dependent effects
Vector3 viewDir = viewerPos.subtract(point).normalize();
double viewAngle = Math.abs(viewDir.dot(normal));
double viewEffect = Math.pow(viewAngle, 0.7);

if (normalizedPattern < 0.3) {
    // Fjord water background
    double depth = normalizedPattern / 0.3;
    return ColorUtil.blendColors(fjordColor,
ColorUtil.darkenColor(fjordColor, 0.4), depth);
} else if (normalizedPattern < 0.6) {
    // Tulip petals with gradient
    double intensity = (normalizedPattern - 0.3) / 0.3;
    Color gradientTulip = ColorUtil.blendColors(tulipColor,
ColorUtil.lightenColor(tulipColor, 0.3), intensity);
    return ColorUtil.multiplyColors(gradientTulip, Color.WHITE,
viewEffect);
} else if (normalizedPattern < 0.8) {
    // Stem and leaf elements
    return stemColor;
} else {
    // Water reflections and highlights
    double intensity = (normalizedPattern - 0.8) / 0.2;
    Color reflection = ColorUtil.blendColors(fjordColor, Color.WHITE,
intensity * 0.5);
    return ColorUtil.lightenColor(reflection, intensity * 0.7 * viewEffect);
}
}

```

```
@Override  
public double getReflectivity() {  
    return reflectivity;  
}  
  
@Override  
public double getIndexOfRefraction() {  
    return ior;  
}  
  
@Override  
public double getTransparency() {  
    return transparency;  
}  
}
```

```
// ======  
// File: /net/elenamurat/material/DielectricMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
import java.util.ArrayList;  
import java.util.List;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.Light;  
import net.elena.murat.util.ColorUtil;  
  
public class DielectricMaterial implements Material {  
  
    // Material properties  
    private Color diffuseColor;  
    private double indexOfRefraction;  
    private double transparency;
```

```
private double reflectivity;  
  
// Filter colors for interior and exterior  
private Color filterColorInside;  
private Color filterColorOutside;  
  
private double currentReflectivity;  
private double currentTransparency;  
  
// Object transformation matrix  
private Matrix4 objectTransform;  
  
/**  
 * Default constructor with glass-like properties  
 */  
public DielectricMaterial() {  
    this.diffuseColor = new Color(0.9f, 0.9f, 0.9f);  
    this.indexOfRefraction = 1.5;  
    this.transparency = 0.8;  
    this.reflectivity = 0.1;  
    this.filterColorInside = new Color(1.0f, 1.0f, 1.0f);  
    this.filterColorOutside = new Color(1.0f, 1.0f, 1.0f);  
    this.objectTransform = new Matrix4().identity();  
  
    this.currentReflectivity = this.reflectivity;  
    this.currentTransparency = this.transparency;  
}  
  
/**  
 * Constructor with custom parameters  
 */  
public DielectricMaterial(Color diffuseColor, double ior,  
    double transparency, double reflectivity) {  
    this.diffuseColor = diffuseColor;  
    this.indexOfRefraction = ior;  
    this.transparency = transparency;  
    this.reflectivity = reflectivity;  
    this.filterColorInside = new Color(1.0f, 1.0f, 1.0f);  
    this.filterColorOutside = new Color(1.0f, 1.0f, 1.0f);
```

```

        this.objectTransform = new Matrix4().identity();
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPoint) {
        Vector3 lightDir = light.getDirectionTo(point).normalize();
        double diffuseFactor = Math.max(0, normal.dot(lightDir));

        // Calculate Fresnel reflection coefficient (for external use or debugging)
        Vector3 viewDir = viewerPoint.subtract(point).normalize();
        double fresnel = Vector3.calculateFresnel(viewDir, normal, 1.0,
        indexOfRefraction);

        this.currentReflectivity = Math.min(0.95, reflectivity + (fresnel * 0.8));
        this.currentTransparency = Math.max(0.05, transparency * (1.0 - fresnel
        * 0.2));

        // Basic diffuse color
        Color diffuse = ColorUtil.multiplyColor(diffuseColor, diffuseFactor *
        light.getIntensity());

        // Specular component
        Vector3 reflectDir = lightDir.reflect(normal);
        double specularFactor = Math.pow(Math.max(0,
        viewDir.dot(reflectDir)), 32);
        Color specular = ColorUtil.multiplyColor(light.getColor(),
        specularFactor * 0.5);

        // Combine diffuse and specular with light color
        Color c1 = ColorUtil.multiplyColors(light.getColor(), diffuse);
        Color result = ColorUtil.add(c1, specular);

        return ColorUtil.clampColor(result);
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {

```

```
if (tm == null) tm = new Matrix4();
this.objectTransform = tm;
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

@Override
public double getTransparency() {
    return currentTransparency;
}

@Override
public double getReflectivity() {
    return currentReflectivity;
}

// Getters and setters for dielectric properties
public Color getFilterColorInside() { return filterColorInside; }
public Color getFilterColorOutside() { return filterColorOutside; }

public void setFilterColorInside(Color filterColorInside) {
    this.filterColorInside = filterColorInside;
}

public void setFilterColorOutside(Color filterColorOutside) {
    this.filterColorOutside = filterColorOutside;
}

public Color getDiffuseColor() {
    return diffuseColor;
}

public void setDiffuseColor(Color diffuseColor) {
    this.diffuseColor = diffuseColor;
}
```

```
public void setIndexOfRefraction(double indexOfRefraction) {  
    this.indexOfRefraction = indexOfRefraction;  
}  
  
public void setTransparency(double transparency) {  
    this.transparency = transparency;  
}  
  
public void setReflectivity(double reflectivity) {  
    this.reflectivity = reflectivity;  
}  
  
@Override  
public String toString() {  
    return String.format("DielectricMaterial[ior=%.2f, transparency=%.2f,  
reflectivity=%.2f]",  
        indexOfRefraction, transparency, reflectivity);  
}  
  
}
```

```
// ======  
// File: /net/lena/murat/material/GlassMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
import java.util.Random;  
  
import net.elena.murat.light.Light;  
import net.elena.murat.math.*;  
import net.elena.murat.util.ColorUtil;  
  
public class GlassMaterial implements Material {  
    private final Color baseColor;  
    private final double indexOfRefraction;  
    private final double reflectivity;
```

```
private final double transparency;
private final Random random;

private double currentReflectivity;
private double currentTransparency;

public GlassMaterial(Color baseColor, double ior,
    double reflectivity, double transparency) {
    this.baseColor = baseColor;
    this.indexOfRefraction = ior;
    this.reflectivity = reflectivity;
    this.transparency = transparency;
    this.random = new Random();

    this.currentReflectivity = this.reflectivity;
    this.currentTransparency = this.transparency;
}

public GlassMaterial(Color baseColor, double ior) {
    this(baseColor, ior, 0.08, 0.92);
}

public GlassMaterial(double ior) {
    this(new Color(200, 220, 240), ior);
}

public GlassMaterial() {
    this(1.5);
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    Vector3 viewDir = viewerPos.subtract(point).normalize();
    Vector3 lightDir = light.getDirectionTo(point).normalize();

    double fresnel = Vector3.calculateFresnel(viewDir, normal, 1.0,
indexOfRefraction);
```

```
this.currentReflectivity = Math.min(0.95, reflectivity + (fresnel * 0.8));
this.currentTransparency = Math.max(0.05, transparency * (1.0 - fresnel
* 0.2));
```

```
double NdotL = Math.max(0.3, normal.dot(lightDir));
double intensity = light.getIntensityAt(point);
```

```
Vector3 reflectDir = lightDir.reflect(normal);
double specular = Math.pow(Math.max(0.0, viewDir.dot(reflectDir)),
128);
```

```
Color glassTint = ColorUtil.multiplyColor(baseColor, 0.6);
Color diffuse = ColorUtil.multiplyColor(glassTint, NdotL * 0.4 *
intensity);
```

```
Color specularHighlight = ColorUtil.multiplyColor(light.getColor(),
specular * 1.2 * intensity);
```

```
Color result = ColorUtil.addSafe(diffuse, specularHighlight);
return ColorUtil.clampColor(result);
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
}
```

```
@Override
public double getReflectivity() {
    return currentReflectivity;
}
```

```
@Override
public double getTransparency() {
    return currentTransparency;
}
```

```
@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}
```

```
public Color getColorForRefraction() {
    return ColorUtil.multiplyColor(baseColor, 0.8);
}

public Color getGlassColor() {
    return baseColor;
}

}

// =====
// File: /net/elenamurat/material/BrunostCheeseMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class BrunostCheeseMaterial implements Material {
    private final Color cheeseColor;
    private final Color caramelColor;
    private final double caramelAmount;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.5;
    private final double diffuseCoeff = 0.9;
    private final double specularCoeff = 0.25;
    private final double shininess = 30.0;
    private final double reflectivity = 0.15;
    private final double ior = 1.55;
    private final double transparency = 0.0;

    public BrunostCheeseMaterial() {
```

```

        this(new Color(0xD2, 0x69, 0x1E), new Color(0x8B, 0x45, 0x13), 0.4);
    }

    public BrunostCheeseMaterial(Color cheeseColor, Color caramelColor,
double caramelAmount) {
    this.cheeseColor = cheeseColor;
    this.caramelColor = caramelColor;
    this.caramelAmount = Math.max(0, Math.min(1, caramelAmount));
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateCheeseTexture(objectPoint);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);
}

```

```

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(new Color(0xFF, 0xEC,
0x8B), props.color, specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateCheeseTexture(Point3 point) {
    double x = point.x * 8.0;
    double y = point.y * 8.0;
    double z = point.z * 8.0;

    // Cheese texture with caramel veins
    double cheeseBase = Math.sin(x * 1.2 + Math.cos(y * 0.8)) * 0.5 + 0.5;
    double caramelVein1 = Math.abs(Math.sin(x * 3.0 + y * 2.0 + z * 1.5));
    double caramelVein2 = Math.abs(Math.cos(x * 2.5 + y * 1.8 + z * 2.0));
    double textureNoise = Math.sin(x * 5.0 + y * 4.0 + z * 3.0) * 0.2 + 0.8;

    double veinPattern = (caramelVein1 * 0.6 + caramelVein2 * 0.4);

    if (veinPattern < caramelAmount) {
        // Caramel veins
        double intensity = veinPattern / caramelAmount;
        Color veinColor = ColorUtil.blendColors(caramelColor,
        ColorUtil.darkenColor(caramelColor, 0.3), intensity);
        return ColorUtil.addColorVariation(veinColor, intensity);
    } else {
        // Cheese base with texture
        double intensity = (veinPattern - caramelAmount) / (1.0 -
caramelAmount);
        Color texturedCheese = ColorUtil.multiplyColors(cheeseColor,
        ColorUtil.createColor(255, 255, 255), textureNoise);
        return ColorUtil.blendColors(texturedCheese,
        ColorUtil.lightenColor(cheeseColor, 0.1), intensity);
    }
}

```

```
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/elenamurat/material/SquaredMaterial.java
// =====

package net.elena.murat.material;

import java.lang.reflect.Method;

import java.awt.Color;

//custom imports
import net.elena.murat.light.*;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

/**
 * SquaredMaterial represents a surface with a two-color square pattern.
 * It now fully implements the extended Material interface, including
 * properties
```

```

* for reflectivity, index of refraction, and transparency.
*/
public class SquaredMaterial implements Material {

    private final Color color1;
    private final Color color2;
    private final double scale; // Represents the frequency of squares (squares
    per unit)
    private Matrix4 objectInverseTransform; // The inverse transformation
matrix of the object

    // Phong lighting model parameters
    private final double ambient;
    private final double diffuse;
    private final double specular;
    private final double shininess;
    private final Color specularColor;
    private final double reflectivity;
    private final double ior;
    private final double transparency;

    /**
     * Constructs a SquaredMaterial with two colors, a scale, Phong lighting
model parameters,
     * and the object's inverse transformation matrix.
     * @param color1 The first color for the square pattern.
     * @param color2 The second color for the square pattern.
     * @param scale The frequency of the squares (e.g., 4.0 for 4 squares per
unit length).
     * @param ambient The ambient reflection coefficient (0.0-1.0).
     * @param diffuse The diffuse reflection coefficient (0.0-1.0).
     * @param specular The specular reflection coefficient (0.0-1.0).
     * @param shininess The shininess exponent for specular highlights.
     * @param specularColor The color of the specular highlight.
     * @param reflectivity The reflectivity coefficient (0.0-1.0).
     * @param ior The Index of Refraction for transparent materials.
     * @param transparency The transparency coefficient (0.0-1.0).
     * @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.

```

```

*/
public SquaredMaterial(Color color1, Color color2, double scale,
    double ambient, double diffuse, double specular,
    double shininess, Color specularColor,
    double reflectivity, double ior, double transparency,
    Matrix4 objectInverseTransform) {
    this.color1 = color1;
    this.color2 = color2;
    this.scale = scale;
    this.objectInverseTransform = objectInverseTransform;

    this.ambient = ambient;
    this.diffuse = diffuse;
    this.specular = specular;
    this.shininess = shininess;
    this.specularColor = specularColor;
    this.reflectivity = reflectivity;
    this.ior = ior;
    this.transparency = transparency;
}

/**
 * Simplified constructor for basic squared pattern without full Phong
parameters.
 * Uses default Phong values.
 * @param color1 The first color for the square pattern.
 * @param color2 The second color for the square pattern.
 * @param scale The frequency of the squares.
 * @param objectInverseTransform The full inverse transformation
matrix of the object.
*/
public SquaredMaterial(Color color1, Color color2, double scale,
    Matrix4 objectInverseTransform) {
    this(color1, color2, scale,
        0.1, 0.7, 0.8, 50.0, Color.WHITE,
        0.0, 1.0, 0.0, objectInverseTransform);
}

@Override

```

```

public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

/**
 * Calculates the color contribution of a single light source at a given
point on the surface,
 * taking into account surface properties and a square pattern using the
Phong model.
 * The hit point is transformed into the material's local space before
pattern calculation.
 *
 * @param worldPoint The point in 3D space (world coordinates) where
the light hits.
 * @param worldNormal The normal vector at the point (world
coordinates).
 * @param light The single light source affecting this point.
 * @param viewerPos The position of the viewer/camera.
 * @return The color contribution from this specific light for the point.
*/
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal,
Light light, Point3 viewerPos) {
// Check if inverse transform is valid before proceeding
if (objectInverseTransform == null) {
    System.err.println("Error: SquaredMaterial's inverse transform is null.
Returning black.");
    return Color.BLACK;
}

// 1. Transform point to object's local space
Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);

// 2. Transform the world normal to local space to determine the local
face orientation
// Normals transform with the inverse transpose of the model matrix.
Vector3 localNormal =

```

```

objectInverseTransform.inverseTransposeForNormal().transformVector(worldNormal).normalize();

    // Check if the transformed normal is valid
    if (localNormal == null) {
        System.err.println("Error: SquaredMaterial's normal transform matrix is null or invalid. Returning black.");
        return Color.BLACK;
    }

    Color patternColor;
    double u, v; // 2D texture coordinates

    // Determine the dominant axis of the *local normal* to decide 2D projection for pattern.
    // This ensures the square pattern aligns correctly with the object's local faces.
    double absNx = Math.abs(localNormal.x);
    double absNy = Math.abs(localNormal.y);
    double absNz = Math.abs(localNormal.z);

    // Project the 3D local point onto a 2D plane based on the dominant local normal axis.
    // Normalize coordinates from [-0.5, 0.5] to [0, 1] for a unit cube local space.
    // Then scale by 'this.scale' which represents squares per unit length.
    // Add a small epsilon to the coordinates before flooring to handle floating point inaccuracies at boundaries.
    if (absNx > absNy && absNx > absNz) { // Normal is mostly X-axis (local Y-Z plane)
        u = (localPoint.y + 0.5 + Ray.EPSILON) * this.scale;
        v = (localPoint.z + 0.5 + Ray.EPSILON) * this.scale;
    } else if (absNy > absNx && absNy > absNz) { // Normal is mostly Y-axis (local X-Z plane)
        u = (localPoint.x + 0.5 + Ray.EPSILON) * this.scale;
        v = (localPoint.z + 0.5 + Ray.EPSILON) * this.scale;
    } else { // Normal is mostly Z-axis (local X-Y plane) or equally dominant
        u = (localPoint.x + 0.5 + Ray.EPSILON) * this.scale;
        v = (localPoint.y + 0.5 + Ray.EPSILON) * this.scale;
    }
}

```

```

}

// Use 2D square pattern logic for all surfaces.
// The parity of the sum of the integer parts determines the color.
int checkU = (int) Math.floor(u);
int checkV = (int) Math.floor(v);

if ((checkU + checkV) % 2 == 0) { // Standard 2D checkerboard pattern
    patternColor = this.color1;
} else {
    patternColor = this.color2;
}

// 3. Phong lighting calculations
Color lightColor = light.getColor();
double attenuatedIntensity = 0.0; // Initialize for non-ambient lights

// Ambient component
Color ambientColor = ColorUtil.multiplyColors(patternColor,
lightColor, ambient);

if (light instanceof ElenaMuratAmbientLight) {
    return ambientColor;
}

// Light direction calculation
Vector3 lightDir = getLightDirection(light, worldPoint);
if (lightDir == null) return Color.BLACK;

// Get attenuated intensity based on light type
if (light instanceof MuratPointLight) {
    attenuatedIntensity = ((MuratPointLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof ElenaDirectionalLight) {
    attenuatedIntensity = ((ElenaDirectionalLight) light).getIntensity();
} else if (light instanceof PulsatingPointLight) {
    attenuatedIntensity = ((PulsatingPointLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof SpotLight) {
}

```

```

attenuatedIntensity = ((SpotLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof BioluminescentLight) {
attenuatedIntensity = ((BioluminescentLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof BlackHoleLight) {
attenuatedIntensity = ((BlackHoleLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof FractalLight) {
attenuatedIntensity = ((FractalLight)
light).getAttenuatedIntensity(worldPoint);
} else {
// Bu else bloguna düşmemesi gerekiyor, çünkü getLightDirection
zaten kontrol ediyor.
System.err.println("Warning: Unknown or unsupported light type for
SquaredMaterial shading (intensity): " + light.getClass().getName());
return Color.BLACK;
}

// Diffuse component
double NdotL = Math.max(0, worldNormal.dot(lightDir));
Color diffuseColor = ColorUtil.multiplyColors(patternColor, lightColor,
diffuse * NdotL * attenuatedIntensity);

// Specular component
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);
Color specularColor = ColorUtil.multiplyColors(this.specularColor,
lightColor, specular * specFactor * attenuatedIntensity);

// Combine all components
return ColorUtil.combineColors(ambientColor, diffuseColor,
specularColor);
}

private Vector3 getLightDirection(Light light, Point3 point) {
if (light == null) {

```

```

        return new Vector3(0, 1, 0).normalize() // Varsayılan yön
    }

    if (light instanceof MuratPointLight) {
        return
        ((MuratPointLight)light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return
        ((ElenaDirectionalLight)light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return
        ((PulsatingPointLight)light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight)light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight)light).getDirectionAt(point);
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight)light).getDirectionAt(point);
    } else if (light instanceof FractalLight) {
        return ((FractalLight)light).getDirectionAt(point);
    } else {
        //return new Vector3(0, 1, 0).normalize();
    }

    // Reflection fallback
    try {
        Method getDirMethod = light.getClass().getMethod("getDirectionAt",
Point3.class);
        return (Vector3) getDirMethod.invoke(light, point);
    }
    catch (Exception e) {
        System.err.println("Unsupported light type: " +
light.getClass().getName());
        return new Vector3(0, 1, 0).normalize(); // Güvenli varsayılan
    }
}

/***
 * Returns the reflectivity coefficient of the material.

```

```

    * @return The reflectivity value (0.0-1.0).
    */
@Override
public double getReflectivity() { return reflectivity; }

/***
 * Returns the index of refraction (IOR) of the material.
 * @return The index of refraction.
 */
@Override
public double getIndexOfRefraction() { return ior; }

/***
 * Returns the transparency coefficient of the material.
 * @return The transparency value (0.0-1.0).
 */
@Override
public double getTransparency() { return transparency; }

}

```

```

// =====
// File: /net/lena/murat/material/CopperMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

public class CopperMaterial extends MetallicMaterial {
    public CopperMaterial() {
        // Copper color (reddish-brown)
        // Reflectivity, shininess and other coefficients are specifically tuned for
        copper
        super(new Color(184, 115, 51), // Copper color (reddish-brown)
              new Color(220, 150, 100), // Slightly reddish specular color for copper
              (or Color.WHITE)
              0.85, // Reflectivity strength (can be slightly less shiny than gold and

```

```
silver)
    120.0, // Shininess (medium-high)
    0.1, // Ambient light contribution
    0.07, // Low diffuse contribution
    0.90 // High specular contribution
);
}

}

// =====
// File: /net/elenamurat/material/SphereWordTextureMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.*;
import java.awt.geom.Point2D;
import java.awt.image.BufferedImage;

import net.elena.murat.math.*;

/**
 * Material that applies a spherical texture with a word drawn on it.
 * The word is rendered on a texture mapped onto a sphere.
 * Supports transparency, reflectivity, gradient text, and background image
parameters.
 */
public class SphereWordTextureMaterial implements Material {
    private final String word;
    private final Color textColor;
    private final Color gradientColor;
    private final String gradientType;
    private final Color bgColor;
    private final String fontFamily;
    private final int fontStyle;
    private final int fontSize;
    private final int uOffset;
```

```

private final int vOffset;
private final BufferedImage imageObject;
private final int imageWidth;
private final int imageHeight;
private final int imageUOffset;
private final int imageVOffset;

private final double reflectivity;
private final double ior;
private final double transparency;

private Matrix4 objectInverseTransform;
private BufferedImage texture;

/**
 * Constructor with default background color (transparent black),
 * reflectivity=0.3, ior=1.0, transparency=0.0 (opaque), no offsets.
 * No gradient (uses solid textColor) and no background image.
 */
public SphereWordTextureMaterial(String word, Color textColor,
        String fontFamily, int fontStyle, int fontSize) {
    this(word, textColor, null, "horizontal", new Color(0x00000000),
fontFamily, fontStyle, fontSize,
        0.3, 1.0, 0.0, 0, 0, null, 0, 0, 0, 0);
}

/**
 * Full constructor with all parameters.
 *
 * @param word      The word to render on the sphere texture.
 * @param textColor Primary text color (also used for gradient
start/end depending on type).
 * @param gradientColor Secondary color for gradient effect. If null,
solid textColor is used.
 * @param gradientType Type of gradient: "horizontal", "vertical",
"diagonal".
 * @param bgColor   Background color of the texture.
 * @param fontFamily Font family name.
 * @param fontStyle Font style (Font.PLAIN, Font.BOLD, etc).

```

```

* @param fontSize    Font size in points.
* @param reflectivity Reflectivity coefficient [0..1].
* @param ior          Index of refraction (>=1.0).
* @param uOffset      Horizontal pixel offset for text positioning.
* @param vOffset      Vertical pixel offset for text positioning.
* @param imageObject  BufferedImage to draw on the background,
can be null.
* @param imageWidth   Width to draw the image.
* @param imageHeight  Height to draw the image.
* @param imageUOffset Horizontal pixel offset for image positioning.
* @param imageVOffset Vertical pixel offset for image positioning.
*/
public SphereWordTextureMaterial(String word, Color textColor, Color
gradientColor,
String gradientType, Color bgColor,
String fontFamily, int fontStyle, int fontSize,
double reflectivity, double ior, double transparency,
int uOffset, int vOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset) {
// Convert special English sequences to Norwegian characters and
replace underscores with spaces
this.word = (convertToNorwegianText(word)).replaceAll("_", " ");
this.textColor = textColor;
this.gradientColor = gradientColor;
this.gradientType = (gradientType != null) ? gradientType :
"horizontal";
this.bgColor = bgColor;
this.fontFamily = fontFamily.replaceAll("_", " ");
this.fontStyle = fontStyle;
this.fontSize = fontSize;
this.reflectivity = Math.min(1.0, Math.max(0.0, reflectivity));
this.ior = Math.max(1.0, ior);
this.transparency=transparency;

this.uOffset = uOffset;
this.vOffset = vOffset;
this.imageObject = imageObject;
this.imageWidth = imageWidth;

```

```
this.imageHeight = imageHeight;
this.imageUOffset = imageUOffset;
this.imageVOffset = imageVOffset;

this.objectInverseTransform = new Matrix4();

this.texture = createTexture();
}

/***
 * Creates the texture image with the word drawn centered, optionally
 * with a gradient and background image.
 * The texture size is fixed at 1024x1024 pixels.
 *
 * @return BufferedImage containing the rendered word texture.
 */
private BufferedImage createTexture() {
    final int size = 1024;
    BufferedImage texture = new BufferedImage(size, size,
BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = texture.createGraphics();

    // Alpha composite setting for proper alpha channel handling
    //g2d.setComposite(AlphaComposite.SrcOver);

    // Enable anti-aliasing for smooth text and images
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
RenderingHints.VALUE_INTERPOLATION_BILINEAR);

    //g2d.setBackground(new Color(0, 0, 0, 0));
    //g2d.clearRect(0, 0, size, size);

    // Set background color with original alpha (transparency applied later in
    //getColorAt)
    g2d.setBackground(bgColor);
```

```

g2d.clearRect(0, 0, size, size);

// Draw background image if provided
if (imageObject != null) {
    int x = (size - imageWidth) / 2 + imageUOffset;
    int y = (size - imageHeight) / 2 - fontStyle / 2;
    y -= (imageHeight/3);
    y += imageVOffset;
    g2d.drawImage(imageObject, x, y, imageWidth, imageHeight, null);
}

// Set font
Font font = new Font(fontFamily, fontStyle, fontSize);
g2d.setFont(font);

// Calculate text position to center it, applying uOffset and vOffset
FontMetrics fm = g2d.getFontMetrics();
int textWidth = fm.stringWidth(word);
int textHeight = fm.getHeight();
int ascent = fm.getAscent();
int descent = fm.getDescent ();

int x = (size - textWidth) / 2 + uOffset;
int y = (size - textHeight) / 2 + (textHeight/3) + ascent + vOffset;

// GradientPaint
if (gradientColor != null && gradientType != null) {
    java.awt.geom.Rectangle2D textBounds = fm.getStringBounds(word,
g2d);

    float textX = x;
    float textY = y - fm.getAscent();

    float ftextWidth = (float) textBounds.getWidth();
    float ftextHeight = (float) textBounds.getHeight();

    GradientPaint gradient;

    switch (gradientType.toLowerCase()) {

```

```
        case "vertical":  
            gradient = new GradientPaint(  
                textX, textY, textColor,  
                textX, textY + ftextHeight/2, gradientColor,  
                true  
            );  
            break;  
  
        case "diagonal":  
            gradient = new GradientPaint(  
                textX, textY, textColor,  
                textX + ftextWidth/3, textY + ftextHeight/5, gradientColor,  
                true  
            );  
            break;  
  
        case "horizontal":  
        default:  
            gradient = new GradientPaint(  
                textX, textY, textColor,  
                textX + ftextWidth/3, textY, gradientColor,  
                true  
            );  
            break;  
        }  
  
        g2d.setPaint(gradient);  
    } else {  
        g2d.setColor(textColor);  
    }  
  
    g2d.drawString(word, x, y);  
    g2d.dispose();  
  
    return texture;  
}  
  
/**  
 * Converts English character sequences to Norwegian special characters.  
 */
```

```

* For example, "AE" -> "Æ", "O/" -> "Ø", "A0" -> "Å", etc.
*
* @param input Input string possibly containing English sequences.
* @return Converted string with Norwegian characters.
*/
public static String convertToNorwegianText(String input) {
    if (input == null || input.isEmpty()) {
        return input;
    }

    String result = input;
    result = result.replace("AE", "\u00C6");
    result = result.replace("O/", "\u00D8");
    result = result.replace("A0", "\u00C5");
    result = result.replace("ae", "\u00E6");
    result = result.replace("o/", "\u00F8");
    result = result.replace("a0", "\u00E5");

    return result;
}

/**
 * Sets the inverse transform matrix of the object.
 * This matrix is used to convert world coordinates to local object
coordinates.
*
* Note: The method keeps the original implementation as requested.
*
* @param tm The transformation matrix of the object.
*/
@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4();
    this.objectInverseTransform = tm;
}

/**
 * Retrieves the color from the texture at the given local 3D point on the
sphere.

```

```

    * Converts the 3D point to spherical coordinates and maps to 2D texture
coordinates.

    *
    * @param localPoint Point in local object coordinates.
    * @return Color from the texture at the mapped UV coordinates.
    */

private Color getTextureColor(Point3 localPoint) {
    if (texture == null) return textColor;

    // Normalize the point to get direction vector on unit sphere
    Vector3 dir = new Vector3(localPoint.x, localPoint.y,
localPoint.z).normalize();

    // Spherical coordinates
    double phi = Math.atan2(dir.z, dir.x); // azimuth angle [-pi, pi]
    double theta = Math.asin(dir.y); // elevation angle [-pi/2, pi/2]

    // Convert spherical coordinates to UV texture coordinates [0..1]
    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI);
    double v = (theta + Math.PI / 2) / Math.PI;
    v = 1.0 - v; // Flip vertically to match texture orientation

    // Apply fixed offset to u to align texture (can be adjusted or
parameterized)
    u = (u + 0.25) % 1.0;

    // Convert UV to pixel coordinates
    int texX = (int) (u * texture.getWidth());
    texX = texX % texture.getWidth();
    if (texX < 0) texX += texture.getWidth();

    int texY = (int) (v * texture.getHeight());
    if (texY < 0 || texY >= texture.getHeight()) {
        // Outside texture bounds, return fully transparent
        return new Color(0, 0, 0, 0);
    }

    return new Color(texture.getRGB(texX, texY), true);
}

```

```

/***
 * Returns the color of the material at the given world point, considering
lighting and viewer position.
 * Applies diffuse and specular lighting based on reflectivity and
transparency.
 *
 * @param worldPoint Point in world coordinates.
 * @param worldNormal Surface normal at the point.
 * @param light Light source affecting the point.
 * @param viewerPos Position of the viewer/camera.
 * @return Color of the material at the point.
*/
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal,
net.elena.murat.light.Light light, Point3 viewerPos) {
    Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
    Color textureColor = getTextureColor(localPoint);

    if (light instanceof net.elena.murat.light.ElenaMuratAmbientLight) {
        return textureColor;
    }

    Vector3 lightDir = getLightDirection(light, worldPoint);
    if (lightDir != null) {
        double diffuseFactor = Math.max(0, worldNormal.dot(lightDir));

        double specularFactor = 0.0;
        if (reflectivity > 0.0) {
            Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
            Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
            specularFactor = Math.pow(Math.max(0, viewDir.dot(reflectDir)),
32) * reflectivity;
        }

        double totalFactor = Math.min(1.0, diffuseFactor + specularFactor);

        return new Color(

```

```

        (int)(textureColor.getRed() * totalFactor),
        (int)(textureColor.getGreen() * totalFactor),
        (int)(textureColor.getBlue() * totalFactor),
        textureColor.getAlpha()
    );
}

return textureColor;
}

/***
 * Helper method to get the light direction vector for various light types.
 *
 * @param light Light source.
 * @param worldPoint Point on the surface in world coordinates.
 * @return Normalized direction vector from point to light or light
direction.
*/
private Vector3 getLightDirection(net.elena.murat.light.Light light,
Point3 worldPoint) {
    if (light instanceof net.elena.murat.light.ElenaDirectionalLight) {
        return ((net.elena.murat.light.ElenaDirectionalLight)
light).getDirection().normalize();
    } else if (light instanceof net.elena.murat.light.MuratPointLight) {
        return ((net.elena.murat.light.MuratPointLight)
light).getPosition().subtract(worldPoint).normalize();
    } else if (light instanceof net.elena.murat.light.PulsatingPointLight) {
        return ((net.elena.murat.light.PulsatingPointLight)
light).getPosition().subtract(worldPoint).normalize();
    } else if (light instanceof net.elena.murat.light.BioluminescentLight) {
        return ((net.elena.murat.light.BioluminescentLight)
light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof net.elena.murat.light.BlackHoleLight) {
        return ((net.elena.murat.light.BlackHoleLight)
light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof net.elena.murat.light.FractalLight) {
        return ((net.elena.murat.light.FractalLight)
light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof net.elena.murat.light.SpotLight) {

```

```
        return ((net.elena.murat.light.SpotLight)
light).getDirectionAt(worldPoint).normalize0;
    }
    return null;
}

/***
 * Returns the reflectivity coefficient of the material.
 *
 * @return Reflectivity [0..1].
 */
@Override
public double getReflectivity() {
    return reflectivity;
}

/***
 * Returns the index of refraction of the material.
 *
 * @return Index of refraction (>=1.0).
 */
@Override
public double getIndexOfRefraction() {
    return ior;
}

/***
 * Returns the transparency coefficient of the material.
 *
 * @return Transparency [0..1], 1.0 = fully transparent.
 */
@Override
public double getTransparency() {
    return transparency;
}

}
```

```
// =====
// File: /net/elenamurat/material/RuneStoneMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class RuneStoneMaterial implements Material {
    private final Color stoneColor;
    private final Color runeColor;
    private final double runeDensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.4;
    private final double diffuseCoeff = 0.75;
    private final double specularCoeff = 0.12;
    private final double shininess = 12.0;
    private final double reflectivity = 0.06;
    private final double ior = 1.8;
    private final double transparency = 0.0;

    public RuneStoneMaterial() {
        this(new Color(0x60, 0x60, 0x60), new Color(0xE8, 0xD8, 0xC8), 0.3);
    }

    public RuneStoneMaterial(Color stoneColor, Color runeColor, double
runeDensity) {
        this.stoneColor = stoneColor;
        this.runeColor = runeColor;
        this.runeDensity = Math.max(0, Math.min(1, runeDensity));
        this.objectTransform = Matrix4.identity();
    }

    @Override
```

```

public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateRunePattern(objectPoint);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(new Color(0xCC, 0xCC,
0xCC), props.color, specularCoeff * specFactor);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateRunePattern(Point3 point) {

```

```

double x = point.x * 8.0;
double y = point.y * 8.0;
double z = point.z * 8.0;

// Create rune-like angular patterns
double pattern1 = Math.abs(Math.sin(x * 1.5) * Math.cos(y * 2.0));
double pattern2 = Math.abs(Math.sin(x * 3.0 + y * 1.7) + Math.cos(y *
2.3 + z * 1.2));
double pattern3 = (Math.floor(x * 0.7) + Math.floor(y * 0.7)) % 2.0;

double combinedPattern = (pattern1 * 0.4 + pattern2 * 0.3 + pattern3 * 0.3);
double normalizedPattern = (combinedPattern % 1.0 + 1.0) % 1.0;

if (normalizedPattern < runeDensity) {
    // Rune carving effect - slightly recessed
    double depth = normalizedPattern / runeDensity;
    return ColorUtil.darkenColor(runeColor, depth * 0.3);
} else {
    // Stone surface
    double variation = (normalizedPattern - runeDensity) / (1.0 -
runeDensity);
    return addStoneTexture(stoneColor, variation);
}
}

```

```

private Color addStoneTexture(Color baseColor, double variation) {
    double noise = Math.sin(variation * 20.0) * 0.1 + 0.9;
    int r = (int)(baseColor.getRed() * noise);
    int g = (int)(baseColor.getGreen() * noise);
    int b = (int)(baseColor.getBlue() * noise);
    return new Color(r, g, b);
}

```

```

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```
@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/elenamurat/material/CrystalClearMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CrystalClearMaterial implements Material {
    private final Color glassTint;
    private final double clarity;
    private final double ior;
    private final double dispersion;

    private Matrix4 objectInverseTransform;

    private static final double BASE_REFLECTIVITY = 0.05;
    private final double reflectivity=BASE_REFLECTIVITY;
    private static final double FRESNEL_BIAS = 0.1;

    public CrystalClearMaterial(Color glassTint, double clarity,
        double ior, double dispersion,
```

```

Matrix4 objectInverseTransform) {
    this.glassTint = glassTint;
    this.clarity = Math.max(0, Math.min(1, clarity));
    this.ior = Math.max(1.3, Math.min(2.0, ior));
    this.dispersion = Math.max(0, Math.min(0.1, dispersion));
    this.objectInverseTransform = objectInverseTransform;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // 1. Light properties getting
    LightProperties lightProps = LightProperties.getLightProperties(light,
worldPoint);
    if (lightProps == null) return glassTint != null ? glassTint :
Color.WHITE;

    // 2. Fresnel
    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    double fresnel = calculateFresnel(viewDir, worldNormal);

    // 3. Colors
    Color refractedColor = calculateDispersion(viewDir, worldNormal,
lightProps);

    // 4. Combine optic
    return combineOptics(
        glassTint != null ? glassTint : Color.WHITE,
        refractedColor,
        fresnel,
        worldNormal,
        lightProps
    );
}

```

```

}

private double calculateFresnel(Vector3 viewDir, Vector3 normal) {
    double cosTheta = Math.abs(viewDir.dot(normal));
    return FRESNEL_BIAS + (1-FRESNEL_BIAS) * Math.pow(1 - cosTheta, 5);
}

private Color calculateDispersion(Vector3 viewDir, Vector3 normal,
LightProperties light) {
    if (dispersion <= 0) return light.color;

    Vector3 refractedR = refract(viewDir, normal, 1.0, ior + dispersion * 0.1);
    Vector3 refractedG = refract(viewDir, normal, 1.0, ior + dispersion * 0.05);
    Vector3 refractedB = refract(viewDir, normal, 1.0, ior);

    return new Color(
        Math.min(255, (int)(light.color.getRed() * 0.9)),
        Math.min(255, (int)(light.color.getGreen() * 0.95)),
        light.color.getBlue()
    );
}

private Vector3 refract(Vector3 incoming, Vector3 normal, double n1,
double n2) {
    double n = n1 / n2;
    double cosI = -normal.dot(incoming);
    double sinT2 = n * n * (1.0 - cosI * cosI);

    if (sinT2 > 1.0) return null; // Total internal reflection

    double cosT = Math.sqrt(1.0 - sinT2);
    return incoming.multiply(n).add(normal.multiply(n * cosI - cosT));
}

private Color combineOptics(Color base, Color refracted, double fresnel,
Vector3 normal, LightProperties light) {

```

```

// Reflection component
Color reflection = new Color(
    Math.min(255, (int)(255 * fresnel * BASE_REFLECTIVITY)),
    Math.min(255, (int)(255 * fresnel * BASE_REFLECTIVITY)),
    Math.min(255, (int)(255 * fresnel * BASE_REFLECTIVITY)))
);

// Refraction component
Color refraction = ColorUtil.blendColors(base, refracted, clarity);

// Light interaction
double NdotL = Math.max(0, normal.dot(light.direction));

return new Color(
    Math.min(255, (int)(refraction.getRed() * light.intensity * NdotL +
reflection.getRed())),
    Math.min(255, (int)(refraction.getGreen() * light.intensity * NdotL +
reflection.getGreen())),
    Math.min(255, (int)(refraction.getBlue() * light.intensity * NdotL +
reflection.getBlue())))
);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return clarity; }

}

/**
Material simpleGlass = new CrystalClearMaterial(
null,      // Colorless
0.95,     // 95% transparency
1.5,       // Standard glass IOR
0.0,       // No color dispersion
testSphere.getInverseTransform()
);

Material leadCrystal = new CrystalClearMaterial(

```

```
new Color(240, 240, 255), // Light blue tint  
0.99, // Highly transparent  
1.7, // High IOR (lead crystal)  
0.02, // Slight color dispersion  
testSphere.getInverseTransform()  
);
```

```
Material artGlass = new CrystalClearMaterial(  
new Color(200, 230, 255, 150), // Blue-green tint  
0.8, // Frosted appearance  
1.6, // Dense glass  
0.05, // Noticeable color dispersion  
testSphere.getInverseTransform()  
);  
*/
```

```
// ======  
// File: /net/elenamurat/material/TexturedCheckerboardMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
import java.awt.Font;  
import java.awt.FontMetrics;  
import java.awt.GradientPaint;  
import java.awt.Graphics2D;  
import java.awt.RenderingHints;  
import java.awt.image.BufferedImage;  
  
import net.elena.murat.math.Matrix4;  
import net.elena.murat.math.Point3;  
import net.elena.murat.math.Vector3;  
import net.elena.murat.light.Light;  
import net.elena.murat.light.ElenaMuratAmbientLight;  
import net.elena.murat.util.ColorUtil;
```

```
public class TexturedCheckerboardMaterial implements Material {  
  
    private final Color color1;  
    private final Color color2;  
    private final double size;  
  
    private final String text;  
    private final Color textColor;  
    private final Color gradientColor;  
    private final String gradientType;  
    private final Color bgColor;  
    private final String fontFamily;  
    private final int fontStyle;  
    private final int fontSize;  
    private final int textUOffset;  
    private final int textVOffset;  
  
    private final BufferedImage imageObject;  
    private final int imageWidth;  
    private final int imageHeight;  
    private final int imageUOffset;  
    private final int imageVOffset;  
  
    private final double ambientCoeff;  
    private final double diffuseCoeff;  
    private final double specularCoeff;  
    private final double shininess;  
    private final Color specularColor;  
  
    private final double reflectivity;  
    private final double ior;  
    private final double transparency;  
  
    private Matrix4 objectInverseTransform;  
    private BufferedImage texture;  
  
    public TexturedCheckerboardMaterial(  
        Color color1, Color color2, double size,  
        String text, Color textColor, Color gradientColor, String gradientType,
```

```
Color bgColor,
String fontFamily, int fontStyle, int fontSize,
int textUOffset, int textVOffset,
BufferedImage imageObject, int imageWidth, int imageHeight,
int imageUOffset, int imageVOffset,
double ambientCoeff, double diffuseCoeff, double specularCoeff,
double shininess, Color specularColor,
double reflectivity, double ior, double transparency,
Matrix4 objectInverseTransform) {

    this.color1 = color1;
    this.color2 = color2;
    this.size = size;

    this.text = text != null ? text.replaceAll("_", " ") : null;
    this.textColor = textColor;
    this.gradientColor = gradientColor;
    this.gradientType = gradientType != null ? gradientType.toLowerCase()
        : "horizontal";
    this.bgColor = bgColor != null ? bgColor : new Color(0, 0, 0, 0);

    this.fontFamily = fontFamily != null ? fontFamily.replaceAll("_", " ")
        : "Arial";
    this.fontStyle = fontStyle;
    this.fontSize = fontSize;
    this.textUOffset = textUOffset;
    this.textVOffset = textVOffset;

    this.imageObject = imageObject;
    this.imageWidth = imageWidth;
    this.imageHeight = imageHeight;
    this.imageUOffset = imageUOffset;
    this.imageVOffset = imageVOffset;

    this.ambientCoeff = ambientCoeff;
    this.diffuseCoeff = diffuseCoeff;
    this.specularCoeff = specularCoeff;
    this.shininess = shininess;
    this.specularColor = specularColor;
```

```

this.reflectivity = reflectivity;
this.ior = ior;
this.transparency = transparency;

    this.objectInverseTransform = objectInverseTransform != null ?
objectInverseTransform : new Matrix4();

    this.texture = createTexture();
}

private BufferedImage createTexture() {
    final int TEX_SIZE = 1024;
    BufferedImage img = new BufferedImage(TEX_SIZE, TEX_SIZE,
BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = img.createGraphics();

    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
RenderingHints.VALUE_RENDER_QUALITY);

    g2d.setBackground(bgColor);
    g2d.clearRect(0, 0, TEX_SIZE, TEX_SIZE);

    int cellSize = (int)(TEX_SIZE * size / 10.0);
    if (cellSize < 1) cellSize = 1;

    for (int y = 0; y < TEX_SIZE; y += cellSize) {
        for (int x = 0; x < TEX_SIZE; x += cellSize) {
            boolean useColor1 = ((x / cellSize) + (y / cellSize)) % 2 == 0;
            g2d.setColor(useColor1 ? color1 : color2);
            g2d.fillRect(x, y, cellSize, cellSize);
        }
    }

    if (imageObject != null) {

```

```

        int imgX = ((TEX_SIZE - imageWidth) / 2) + imageUOffset -
imageWidth - (imageWidth / 4);
        int imgY = ((TEX_SIZE - imageHeight) / 2) + imageVOffset -
(imageHeight/2);
        g2d.drawImage(imageObject, imgX, imgY, imageWidth, imageHeight,
null);
    }

if(text != null && !text.trim().isEmpty()) {
    Font font;
    try {
        font = new Font(fontFamily, fontStyle, fontSize);
    } catch (Exception e) {
        font = new Font("Arial", fontStyle, fontSize);
    }
    g2d.setFont(font);

    FontMetrics fm = g2d.getFontMetrics();
    int textWidth = fm.stringWidth(text);
    int textHeight = fm.getHeight();
    int ascent = fm.getAscent();

    int x = ((TEX_SIZE - textWidth) / 2) + textUOffset - textWidth -
(textWidth / 4);
    int y = ((TEX_SIZE - textHeight) / 2) + (ascent * 2) + textVOffset;

    if (gradientColor != null) {
        GradientPaint gradient = createGradient(x, y - ascent, textWidth,
textHeight);
        g2d.setPaint(gradient);
    } else {
        g2d.setColor(textColor);
    }

    g2d.drawString(text, x, y);
}

g2d.dispose();
return img;
}

```

```

}

private GradientPaint createGradient(float x, float y, float width, float
height) {
    switch (gradientType) {
        case "vertical":
            return new GradientPaint(x, y, textColor, x, y + height/2,
gradientColor, true);
        case "diagonal":
            return new GradientPaint(x, y, textColor, x + width/3, y + height/5,
gradientColor, true);
        case "horizontal":
        default:
            return new GradientPaint(x, y, textColor, x + width/3, y,
gradientColor, true);
    }
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if(texture == null) {
        return color1;
    }

    Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    double phi = Math.atan2(localNormal.z, localNormal.x);
    double theta = Math.asin(localNormal.y);

    double u = 1.0 - (phi + Math.PI) / (2 * Math.PI);
    double v = (theta + Math.PI/2) / Math.PI;
    v = 1.0 - v;

    int texX = (int)(u * texture.getWidth()) % texture.getWidth();

```

```

int texY = (int)(v * texture.getHeight());
if (texY < 0) texY = 0;
if (texY >= texture.getHeight()) texY = texture.getHeight() - 1;

Color baseColor = new Color(texture.getRGB(texX, texY), true);

if (baseColor.getAlpha() == 0) {
    double scaledX = localPoint.x * size;
    double scaledY = localPoint.y * size;
    double scaledZ = localPoint.z * size;

    int ix = (int)Math.floor(scaledX);
    int iy = (int)Math.floor(scaledY);
    int iz = (int)Math.floor(scaledZ);

    boolean isColor1 = ((ix + iy + iz) % 2 == 0);
    baseColor = isColor1 ? color1 : color2;
}

if (light == null || light instanceof ElenaMuratAmbientLight) {
    return ColorUtil.multiplyColor(baseColor, ambientCoeff);
}

Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();
double NdotL = Math.max(0, worldNormal.dot(lightDir));

Color ambient = ColorUtil.multiplyColor(baseColor, ambientCoeff);
Color diffuse = ColorUtil.multiplyColors(baseColor, light.getColor(),
diffuseCoeff * NdotL * light.getIntensity());

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = lightDir.reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * light.getIntensity();
Color specular = ColorUtil.multiplyColors(specularColor,
light.getColor(), specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm != null) {
        this.objectInverseTransform = tm;
    }
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public String toString() {
    return "TexturedCheckerboardMaterial[text=" + text + ", size=" + size +
"]";
}

}

// =====
// File: /net/lena/murat/material/RoughMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
```

```

import net.elena.murat.light.*;
import net.elena.murat.math.*;

/**
 * RoughMaterial: Optimized material for rough surfaces.
 * Simulates micro-surface roughness unlike DiffuseMaterial.
 * Simple implementation following Physically Based Rendering (PBR)
principles.
*/
public class RoughMaterial implements Material {
    private final Color color;
    private final double roughness; // 0.0 (smooth) - 1.0 (very rough)
    private final double diffuseCoefficient;
    private final double reflectivity; // 0.0 - 1.0

    /**
     * Full constructor.
     * @param color Material color
     * @param roughness Surface roughness (0.0-1.0)
     * @param diffuseCoefficient Diffuse reflection coefficient (0.0-1.0)
     * @param reflectivity Base reflectivity amount (0.0-1.0)
     */
    public RoughMaterial(Color color, double roughness,
        double diffuseCoefficient, double reflectivity) {
        this.color = color;
        this.roughness = clamp01(roughness);
        this.diffuseCoefficient = clamp01(diffuseCoefficient);
        this.reflectivity = clamp01(reflectivity);
    }

    /**
     * Simple constructor: takes only color and roughness.
     * @param color Material color
     * @param roughness Surface roughness (0.0-1.0)
     */
    public RoughMaterial(Color color, double roughness) {
        this(color, roughness, 0.9, 0.05 * roughness); // Reflectivity decreases as
roughness increases
    }
}

```

```

}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    if (light instanceof ElenaMuratAmbientLight) {
        return applyRoughnessToColor(color);
    }

    Vector3 lightDir = getLightDirection(light, point);
    if (lightDir == null) return Color.BLACK;

    double intensity = getLightIntensity(light, point);
    if (intensity <= 0) return Color.BLACK;

    // Diffuse calculation based on roughness
    double NdotL = Math.max(0.0, normal.dot(lightDir));
    double roughFactor = 1.0 - (roughness * 0.7); // Roughness effect
    double contribution = diffuseCoefficient * NdotL * intensity *
    roughFactor;

    Color baseColor = applyRoughnessToColor(color);
    int r = (int) (baseColor.getRed() * contribution);
    int g = (int) (baseColor.getGreen() * contribution);
    int b = (int) (baseColor.getBlue() * contribution);

    return new Color(
        Math.min(255, Math.max(0, r)),
        Math.min(255, Math.max(0, g)),
        Math.min(255, Math.max(0, b)))
    );
}

/**
 * Darkens color based on roughness (simulates micro-shadows)
 */
private Color applyRoughnessToColor(Color original) {
    float[] hsb = Color.RGBtoHSB(
        original.getRed(),

```

```

        original.getGreen(),
        original.getBlue(),
        null
    );
    // Value (brightness) decreases as roughness increases
    hsb[2] = (float) (hsb[2] * (1.0 - (roughness * 0.3)));
    return new Color(Color.HSBtoRGB(hsb[0], hsb[1], hsb[2]));
}

// --- Material Interface Implementations ---
@Override
public double getReflectivity() {
    return reflectivity * (1.0 - roughness); // Roughness reduces reflectivity
}

@Override
public double getIndexOfRefraction() {
    return 1.0; // IOR = 1.0 for solid opaque materials (no light refraction)
}

@Override
public double getTransparency() {
    return 0.0; // Fully opaque
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

// --- Helper Methods ---

private double getLightIntensity(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight) light).getIntensity();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof SpotLight) {

```

```

        return ((SpotLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getAttenuatedIntensity(point);
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getAttenuatedIntensity(point);
    }
    return 1.0; // Default
}

private Vector3 getLightDirection(Light light, Point3 point) {
    if (light instanceof MuratPointLight) {
        return ((MuratPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return ((ElenaDirectionalLight)
light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return ((PulsatingPointLight)
light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight)
light).getDirectionAt(point).normalize();
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight) light).getDirectionAt(point).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight) light).getDirectionAt(point).normalize();
    }
    return null;
}

private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}

// --- Getters ---

```

```
public Color getColor() {
    return color;
}

public double getRoughness() {
    return roughness;
}

public double getDiffuseCoefficient() {
    return diffuseCoefficient;
}

}

// =====
// File: /net/elenamurat/material/BrightnessMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

public class BrightnessMaterial implements Material {

    private Color baseColor;
    private double brightness;
    private boolean useLightColor;

    private double transparency = 0.0; //opaque

    public BrightnessMaterial(Color baseColor, double brightness) {
        this(baseColor, brightness, false);
    }

    public BrightnessMaterial(Color baseColor, double brightness, boolean
```

```

useLightColor) {
    this.baseColor = baseColor;
    this.brightness = brightness;
    this.useLightColor = useLightColor;
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Color sourceColor = baseColor;

    if (useLightColor) {
        sourceColor = light.getColor();
    }

    int alfa = sourceColor.getAlpha ();
    double alpha = ((double)(alfa))/255.0;

    setTransparency (1-alpha);

    return applyBrightness(sourceColor, brightness);
}

private Color applyBrightness(Color color, double brightnessFactor) {
    float[] rgb = color.getRGBColorComponents(null);

    float r = (float) Math.max(0.0, Math.min(1.0, rgb[0] *
brightnessFactor));
    float g = (float) Math.max(0.0, Math.min(1.0, rgb[1] *
brightnessFactor));
    float b = (float) Math.max(0.0, Math.min(1.0, rgb[2] *
brightnessFactor));

    return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return 0.0;
}

```

```
}
```

```
@Override  
public double getTransparency() {  
    return transparency;  
}
```

```
public void setTransparency (double tnw){  
    this.transparency = tnw;  
}
```

```
@Override  
public double getIndexOfRefraction() {  
    return 1.0;  
}
```

```
@Override  
public void setObjectTransform(Matrix4 tm) {  
}
```

```
// Getters Setters  
public Color getBaseColor() {  
    return baseColor;  
}
```

```
public void setBaseColor(Color baseColor) {  
    this.baseColor = baseColor;  
}
```

```
public double getBrightness() {  
    return brightness;  
}
```

```
public void setBrightness(double brightness) {  
    this.brightness = Math.max(0.0, brightness);  
}
```

```
public boolean isUseLightColor() {  
    return useLightColor;
```

```
}

public void setUseLightColor(boolean useLightColor) {
    this.useLightColor = useLightColor;
}

}

// =====
// File: /net/elenamurat/material/GoldMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

public class GoldMaterial extends MetallicMaterial {
    public GoldMaterial() {
        super(new Color(255, 215, 0), // Gold color
              new Color(255, 223, 186), // Light yellowish specular color for gold
              (or Color.WHITE)
              0.9, // Reflectivity strength
              150.0, // Shininess
              0.1, // Ambient light contribution
              0.05, // Very low diffuse contribution
              0.95 // High specular contribution
        );
    }
}

// =====
// File: /net/elenamurat/material/RandomMaterial.java
// =====

package net.elena.murat.material;
```

```

import java.awt.Color;
import java.util.Random;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

public class RandomMaterial implements Material {
    private static final Random RAND = new Random();
    private final Color diffuseColor;
    private final Color specularColor;
    private final double ambientCoeff;
    private final double diffuseCoeff;
    private final double specularCoeff;
    private final double shininess;
    private final double reflectivity;
    private final double transparency;
    private Matrix4 objectInverseTransform;

    public RandomMaterial(Matrix4 invTransform) {
        this.diffuseColor = randomColor();
        this.specularColor = randomColor();
        this.ambientCoeff = randomInRange(0.1, 0.3);
        this.diffuseCoeff = randomInRange(0.5, 1.0);
        this.specularCoeff = randomInRange(0.1, 0.9);
        this.shininess = randomInRange(5, 150);
        this.reflectivity = randomInRange(0, 0.5);
        this.transparency = RAND.nextBoolean() ? randomInRange(0, 0.3) : 0;
        this.objectInverseTransform = invTransform;
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectInverseTransform = tm;
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 normal, Light light,

```

```

Point3 viewPos) {
    // Diffuse
    Vector3 lightDir = light.getPosition().subtract(worldPoint).normalize();
    double NdotL = Math.max(0, normal.dot(lightDir));
    Color diffuse = ColorUtil.multiplyColors(diffuseColor, light.getColor(),
    diffuseCoeff * NdotL);

    // Specular (Blinn-Phong)
    Vector3 viewDir = viewPos.subtract(worldPoint).normalize();
    Vector3 halfway = lightDir.add(viewDir).normalize();
    double specFactor = Math.pow(Math.max(0, normal.dot(halfway)),
    shininess);
    Color specular = ColorUtil.multiplyColors(specularColor,
    light.getColor(), specularCoeff * specFactor);

    // Ambient
    Color ambient = ColorUtil.multiplyColors(diffuseColor,
    light.getColor(), ambientCoeff);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

// Helper methods
private Color randomColor() {
    return new Color(RAND.nextInt(256), RAND.nextInt(256),
    RAND.nextInt(256));
}

private double randomInRange(double min, double max) {
    return min + (max - min) * RAND.nextDouble();
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.0 +
RAND.nextDouble(); }
@Override public double getTransparency() { return transparency; }

}

```

```

/***
// Adding random material into the scene
Sphere sphere = new Sphere(1.0);
sphere.setMaterial(new RandomMaterial(sphere.getInverseTransform()));
scene.addShape(sphere);
*/
// =====
// File: /net/elenamurat/material/OrbitalMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

public class OrbitalMaterial implements Material {
    private Color centerColor;
    private Color orbitColor;
    private double ringWidth;
    private int ringCount;
    private double transparency;

    public OrbitalMaterial(Color centerColor, Color orbitColor,
        double ringWidth, int ringCount) {
        this.centerColor = centerColor;
        this.orbitColor = orbitColor;
        this.ringWidth = ringWidth;
        this.ringCount = ringCount;
        this.transparency = calculateTransparency(centerColor);
    }

    public OrbitalMaterial(Color centerColor, Color orbitColor) {
        this(centerColor, orbitColor, 0.1, 5);
    }
}

```

```
private double calculateTransparency(Color color) {
    int alpha = color.getAlpha();
    return 1.0 - ((double)alpha / 255.0);
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return 0.2;
}

@Override
public double getIndexOfRefraction() {
    return 1.4;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    // Not needed for this material
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    double pattern = calculateOrbitalPattern(point);
    return ColorUtil.blendColors(centerColor, orbitColor, pattern);
}

private double calculateOrbitalPattern(Point3 point) {
    double distanceFromCenter = Math.sqrt(
        point.x * point.x + point.y * point.y + point.z * point.z
    );
    // Calculate ring pattern based on distance
}
```

```
    double ringPattern = Math.sin(distanceFromCenter * ringCount *  
Math.PI);  
  
    // Apply smooth step function for sharp rings  
    double ringValue = smoothStep(0.5 - ringWidth, 0.5 + ringWidth,  
ringPattern * ringPattern);  
  
    return ringValue;  
}  
  
private double smoothStep(double edge0, double edge1, double x) {  
    x = clamp((x - edge0) / (edge1 - edge0), 0.0, 1.0);  
    return x * x * (3.0 - 2.0 * x);  
}  
  
private double clamp(double value, double min, double max) {  
    return Math.max(min, Math.min(max, value));  
}  
  
public Color getCenterColor() {  
    return centerColor;  
}  
  
public void setCenterColor(Color centerColor) {  
    this.centerColor = centerColor;  
    this.transparency = calculateTransparency(centerColor);  
}  
  
public Color getOrbitColor() {  
    return orbitColor;  
}  
  
public void setOrbitColor(Color orbitColor) {  
    this.orbitColor = orbitColor;  
}  
  
public double getRingWidth() {  
    return ringWidth;  
}
```

```
public void setRingWidth(double ringWidth) {
    this.ringWidth = ringWidth;
}

public int getRingCount() {
    return ringCount;
}

public void setRingCount(int ringCount) {
    this.ringCount = ringCount;
}
}
```

```
// =====
// File: /net/elenamurat/material/VikingRuneMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class VikingRuneMaterial implements Material {
    private final Color stoneColor;
    private final Color runeColor;
    private final double runeDepth;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.4;
    private final double diffuseCoeff = 0.75;
    private final double specularCoeff = 0.1;
    private final double shininess = 12.0;
    private final double reflectivity = 0.05;
    private final double ior = 1.8;
```

```

private final double transparency = 0.0;

public VikingRuneMaterial() {
    this(new Color(0x60, 0x60, 0x60), new Color(0xE8, 0xD8, 0xC8),
0.35);
}

public VikingRuneMaterial(Color stoneColor, Color runeColor, double
runeDepth) {
    this.stoneColor = stoneColor;
    this.runeColor = runeColor;
    this.runeDepth = Math.max(0, Math.min(1, runeDepth));
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateRuneCarving(objectPoint, worldNormal);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }
}

```

```

        double NdotL = Math.max(0, worldNormal.dot(props.direction));
        Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

        Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
        Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
        double RdotV = Math.max(0, reflectDir.dot(viewDir));
        double specFactor = Math.pow(RdotV, shininess) * props.intensity;
        Color specular = ColorUtil.multiplyColors(new Color(0xCC, 0xCC,
0xCC), props.color, specularCoeff * specFactor);

        return ColorUtil.combineColors(ambient, diffuse, specular);
    }

private Color calculateRuneCarving(Point3 point, Vector3 normal) {
    double x = point.x * 10.0;
    double y = point.y * 10.0;
    double z = point.z * 10.0;

    // Viking rune angular patterns
    double rune1 = Math.abs(Math.sin(x * 2.5) * Math.cos(y * 2.0));
    double rune2 = Math.abs(Math.sin(x * 3.0 + y * 1.7) + Math.cos(y * 2.3
+ z * 1.2));
    double rune3 = (Math.floor(x * 0.8) + Math.floor(y * 0.8)) % 2.5;
    double ancient = Math.sin(x * 1.2 + y * 0.8 + z * 0.5) * 0.3 + 0.7;

    double combinedPattern = (rune1 * 0.35 + rune2 * 0.3 + rune3 * 0.25 +
ancient * 0.1);
    double normalizedPattern = combinedPattern % 1.0;

    if (normalizedPattern < runeDepth) {
        // Deep rune carvings
        double depth = normalizedPattern / runeDepth;
        return ColorUtil.darkenColor(runeColor, depth * 0.6);
    } else {
        // Weathered stone surface
        double weathering = (normalizedPattern - runeDepth) / (1.0 -
runeDepth);
    }
}

```

```
        Color weatheredStone = ColorUtil.addColorVariation(stoneColor,
weathering);
        return ColorUtil.darkerColor(weatheredStone, weathering * 0.2);
    }
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/elenamurat/material/TransparentEmissivePNGMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.Light;
import net.elena.murat.math.Matrix4;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

/**
```

- * Material that combines transparent PNG texture with emissive properties.

- * The texture provides the base color and alpha channel, while emissive properties

- * add self-illumination effects. Perfect for glowing transparent objects like emojis.

*/

```
public class TransparentEmissivePNGMaterial implements Material {
```

```
    private BufferedImage texture;
```

```
    private Matrix4 objectInverseTransform = new Matrix4();
```

```
    // UV parameters
```

```
    private double uOffset = 0.0;
```

```
    private double vOffset = 0.0;
```

```
    private double uScale = 1.0;
```

```
    private double vScale = 1.0;
```

```
    private boolean isRepeatTexture = false;
```

```
    // Emissive properties
```

```
    private Color emissiveColor;
```

```
    private double emissiveStrength;
```

```
    private double transparency = 1.0;
```

```
/**
```

- * Constructor with texture and emissive properties

- * @param texture BufferedImage with alpha channel (PNG)

- * @param emissiveColor Color of the emitted light

- * @param emissiveStrength Intensity of the emission (0.0 - 1.0 or higher)

*/

```
    public TransparentEmissivePNGMaterial(BufferedImage texture, Color emissiveColor, double emissiveStrength) {
```

```
        this.texture = texture;
```

```
        this.emissiveColor = new Color(
```

```
            emissiveColor.getRed(),
```

```
            emissiveColor.getGreen(),
```

```
            emissiveColor.getBlue()
```

```
        );
```

```

        this.emissiveStrength = Math.max(0, emissiveStrength);
    }

/***
 * Full constructor with UV parameters and emissive properties
 */
public TransparentEmissivePNGMaterial(BufferedImage texture, double
uOffset, double vOffset,
double uScale, double vScale, boolean isRepeatTexture,
Color emissiveColor, double emissiveStrength) {
    this.texture = texture;
    this.uOffset = uOffset;
    this.vOffset = vOffset;
    this.uScale = (uScale > 0.0) ? uScale : 1.0;
    this.vScale = (vScale > 0.0) ? vScale : 1.0;
    this.isRepeatTexture = isRepeatTexture;
    this.emissiveColor = new Color(
        emissiveColor.getRed(),
        emissiveColor.getGreen(),
        emissiveColor.getBlue()
    );
    this.emissiveStrength = Math.max(0, emissiveStrength);
}

```

@Override

```

public void setObjectTransform(Matrix4 inverseTransform) {
    if (inverseTransform != null) {
        this.objectInverseTransform = inverseTransform;
    } else {
        this.objectInverseTransform = new Matrix4();
    }
}

```

@Override

```

public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    if (texture == null) {
        setTransparency(1.0);
        return new Color(0, 0, 0, 0);
    }
}

```

```
}
```

```
Point3 local = objectInverseTransform.transformPoint(point);

// Base UV coordinates mapped from [-1,1] to [0,1]
double u = (local.x + 1.0) * 0.5;
double v = (1.0 - (local.y + 1.0)) * 0.5;

// Apply UV scale and offset
double scaledU = u / uScale + uOffset;
double scaledV = v / vScale + vOffset;

double finalU, finalV;

if (isRepeatTexture) {
    // Wrap UVs for tiling (repeat)
    finalU = scaledU - Math.floor(scaledU);
    finalV = scaledV - Math.floor(scaledV);
} else {
    // No tiling: if UV outside [0,1], return fully transparent color
    if (scaledU < 0.0 || scaledU > 1.0 || scaledV < 0.0 || scaledV > 1.0) {
        setTransparency(1.0);
        return new Color(0, 0, 0, 0);
    }
    finalU = scaledU;
    finalV = scaledV;
}

int px = (int) (finalU * (texture.getWidth() - 1));
int py = (int) (finalV * (texture.getHeight() - 1));

int argb = texture.getRGB(px, py);

int alpha = (argb >> 24) & 0xFF;
int red = (argb >> 16) & 0xFF;
int green = (argb >> 8) & 0xFF;
int blue = argb & 0xFF;

if (alpha > 5) {
```

```

        setTransparency(0.0);

        int emissiveRed = (int) (emissiveColor.getRed() * emissiveStrength);
        int emissiveGreen = (int) (emissiveColor.getGreen() *
emissiveStrength);
        int emissiveBlue = (int) (emissiveColor.getBlue() * emissiveStrength);

        int finalRed = clampColorValue(red + emissiveRed);
        int finalGreen = clampColorValue(green + emissiveGreen);
        int finalBlue = clampColorValue(blue + emissiveBlue);

        return new Color(finalRed, finalGreen, finalBlue, 255);
    }

    setTransparency(1.0);
    return new Color(0, 0, 0, 0);
}

/***
 * Gets the color from the texture at the given point
 */
private Color getTextureColor(Point3 point) {
    if(texture == null) {
        return new Color(0, 0, 0, 0);
    }

    Point3 local = objectInverseTransform.transformPoint(point);
    double u = (local.x + 1.0) * 0.5;
    double v = (1.0 - (local.y + 1.0)) * 0.5;

    double scaledU = u / uScale + uOffset;
    double scaledV = v / vScale + vOffset;

    double finalU, finalV;

    if(isRepeatTexture) {
        finalU = scaledU - Math.floor(scaledU);
        finalV = scaledV - Math.floor(scaledV);
    } else {

```

```

    if (scaledU < 0.0 || scaledU > 1.0 || scaledV < 0.0 || scaledV > 1.0) {
        return new Color(0, 0, 0, 0);
    }
    finalU = scaledU;
    finalV = scaledV;
}

int px = (int) (finalU * (texture.getWidth() - 1));
int py = (int) (finalV * (texture.getHeight() - 1));
int argb = texture.getRGB(px, py);

int alpha = (argb >> 24) & 0xFF;
int red = (argb >> 16) & 0xFF;
int green = (argb >> 8) & 0xFF;
int blue = argb & 0xFF;

return new Color(red, green, blue, alpha);
}

/***
 * Combines texture color with emissive light
 */
private Color combineTextureWithEmission(Color textureColor) {
    // Get texture RGB components
    int r = textureColor.getRed();
    int g = textureColor.getGreen();
    int b = textureColor.getBlue();

    // Get emissive RGB components scaled by strength
    int er = (int) (emissiveColor.getRed() * emissiveStrength);
    int eg = (int) (emissiveColor.getGreen() * emissiveStrength);
    int eb = (int) (emissiveColor.getBlue() * emissiveStrength);

    // Add emissive light to texture color (with clamping)
    int finalR = clampColorValue(r + er);
    int finalG = clampColorValue(g + eg);
    int finalB = clampColorValue(b + eb);

    return new Color(finalR, finalG, finalB, textureColor.getAlpha());
}

```

```
}
```

```
private int clampColorValue(double value) {  
    return (int) Math.min(255, Math.max(0, value));  
}
```

```
@Override  
public double getReflectivity() {  
    return 0.0;  
}
```

```
@Override  
public double getIndexOfRefraction() {  
    return 1.0;  
}
```

```
@Override  
public double getTransparency() {  
    return transparency;  
}
```

```
private void setTransparency (double tnw) {  
    this.transparency = tnw;  
}
```

```
// Getters and setters for emissive properties  
public Color getEmissiveColor() {  
    return new Color(  
        emissiveColor.getRed(),  
        emissiveColor.getGreen(),  
        emissiveColor.getBlue()  
    );  
}
```

```
public void setEmissiveColor(Color emissiveColor) {  
    this.emissiveColor = new Color(  
        emissiveColor.getRed(),  
        emissiveColor.getGreen(),  
        emissiveColor.getBlue()
```

```
        );
    }

    public double getEmissiveStrength() {
        return emissiveStrength;
    }

    public void setEmissiveStrength(double emissiveStrength) {
        this.emissiveStrength = Math.max(0, emissiveStrength);
    }

    // UV parameter getters and setters
    public double getUOffset() { return uOffset; }
    public void setUOffset(double uOffset) { this.uOffset = uOffset; }

    public double getVOffset() { return vOffset; }
    public void setVOffset(double vOffset) { this.vOffset = vOffset; }

    public double getUScale() { return uScale; }
    public void setUScale(double uScale) { this.uScale = uScale > 0.0 ?
        uScale : 1.0; }

    public double getVScale() { return vScale; }
    public void setVScale(double vScale) { this.vScale = vScale > 0.0 ?
        vScale : 1.0; }

    public boolean isRepeatTexture() { return isRepeatTexture; }
    public void setRepeatTexture(boolean repeat) { this.isRepeatTexture =
repeat; }

}
```

```
// =====
// File: /net/elenamurat/material/StarfieldMaterial.java
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class StarfieldMaterial implements Material {
    private Matrix4 objectInverseTransform;
    private final Color nebulaColor;
    private final double starSize; // Range 0.001-0.1
    private final double starDensity;
    private final double twinkleSpeed;
    private final double reflectivity = 0.02;

    // Star color palette
    private static final Color[] STAR_COLORS = {
        new Color(255, 255, 255), // White
        new Color(200, 200, 255), // Blue
        new Color(255, 200, 150), // Yellowish
        new Color(200, 255, 200) // Greenish
    };

    public StarfieldMaterial(Matrix4 objectInverseTransform) {
        this(objectInverseTransform,
            new Color(10, 5, 40), // Nebula color
            0.015, // Star size
            0.003, // Star density
            1.0 // Twinkle speed
        );
    }

    public StarfieldMaterial(Matrix4 objectInverseTransform,
        Color nebulaColor,
        double starSize,
        double starDensity,
        double twinkleSpeed) {
        this.objectInverseTransform = objectInverseTransform;
        this.nebulaColor = nebulaColor;
        this.starSize = Math.min(0.1, Math.max(0.001, starSize));
    }
}
```

```

this.starDensity = Math.min(0.01, Math.max(0.0001, starDensity));
this.twinkleSpeed = Math.max(0.1, twinkleSpeed);
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if(tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 normal, Light light,
Point3 viewPos) {
    Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
    double dist = localPoint.distance(Point3.ORIGIN);

    // Star generation
    double starValue = calculateStarValue(localPoint, dist);

    if(starValue > 0) {
        // Star color and brightness
        Color starColor = getStarColor(localPoint);
        float brightness = (float)(starValue * (0.7 +
0.3*Math.sin(System.currentTimeMillis()*0.001*twinkleSpeed)));

        // Light effect
        double lightEffect = calculateLightEffect(light, worldPoint, normal);
        return ColorUtil.blendColors(starColor, light.getColor(), brightness *
lightEffect);
    }

    // Nebula color (illuminated by light)
    return applyNebulaLight(nebulaColor, light, dist);
}

private double calculateStarValue(Point3 p, double dist) {
    // Make star positions deterministic
    long seed = (long)(p.x*1000) ^ (long)(p.y*1000) ^ (long)(p.z*1000);
}

```

```

double random = (seed * 9301 + 49297) % 233280 / (double)233280;

// Scale by distance
double scaledDensity = starDensity * (1.0 + dist * 0.5);

if(random < scaledDensity) {
    // Calculate star size
    double sizeFactor = 1.0 + (seed % 1000)/1000.0; // Range 1.0-2.0
    return starSize * sizeFactor;
}
return 0;
}

private Color getStarColor(Point3 p) {
    // Deterministic color selection
    int hash = (int)(p.x*100 + p.y*100 + p.z*100) %
STAR_COLORS.length;
    return STAR_COLORS[Math.abs(hash)];
}

private double calculateLightEffect(Light light, Point3 worldPoint,
Vector3 normal) {
    // Light direction and intensity
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();
    double intensity = light.getIntensityAt(worldPoint);
    double NdotL = Math.max(0.1, normal.dot(lightDir));
    return NdotL * intensity;
}

private Color applyNebulaLight(Color nebula, Light light, double dist) {
    // Illuminate nebula with light
    double lightFactor = 0.2 + 0.1 * Math.sin(dist * 0.5) *
light.getIntensity();
    int red = Math.min(255, ((int)(nebula.getRed() * lightFactor)));
    int green = Math.min(255, ((int)(nebula.getGreen() * lightFactor)));
    int blue = Math.min(255, ((int)(nebula.getBlue() * lightFactor)));

    return new Color(red, green, blue);
}

```

```
@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.95; }
}

/***
Material starfield = new StarfieldMaterial(
sphere.getInverseTransform(),
new Color(15, 10, 50), // Nebula color
0.05, // Very large stars
0.002, // More sparse
0.8 // Slow twinkle
);
*/

```

```
// =====
// File: /net/elenamurat/material/RubyMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

public class RubyMaterial implements Material {

    private Color baseColor;
    private double density;
    private double reflectivity;
    private Matrix4 objectTransform;

    public RubyMaterial() {
```

```

this.baseColor = new Color(220, 20, 30);
this.density = 0.8;
this.reflectivity = 0.1;
}

public RubyMaterial(Color baseColor, double density, double
reflectivity) {
    this.baseColor = baseColor;
    this.density = Math.max(0, Math.min(1, density));
    this.reflectivity = Math.max(0, Math.min(1, reflectivity));
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Color lightColor = light.getColor();

    double red = (lightColor.getRed() * baseColor.getRed() / 255.0) *
density;
    double green = (lightColor.getGreen() * baseColor.getGreen() / 255.0) *
density * 0.3; // Yeşili bastır
    double blue = (lightColor.getBlue() * baseColor.getBlue() / 255.0) *
density * 0.2; // Maviyi bastır

    int r = ColorUtil.clampColorValue((int) red);
    int g = ColorUtil.clampColorValue((int) green);
    int b = ColorUtil.clampColorValue((int) blue);

    return new Color(r, g, b);
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

@Override
public double getTransparency() {
    return 0.7;
}

```

```
}

@Override
public double getIndexOfRefraction() {
    return 1.76;
}

@Override
public double getReflectivity() {
    return this.reflectivity;
}

public Color getBaseColor() {
    return baseColor;
}

public void setBaseColor(Color baseColor) {
    this.baseColor = baseColor;
}

public double getDensity() {
    return density;
}

public void setDensity(double density) {
    this.density = Math.max(0, Math.min(1, density));
}

public double getReflectivityValue() {
    return reflectivity;
}

public void setReflectivity(double reflectivity) {
    this.reflectivity = Math.max(0, Math.min(1, reflectivity));
}

}
```

```
// =====
// File: /net/elenamurat/material/CarpetTextureMaterial.java
// =====

// CarpetTextureMaterial.java
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CarpetTextureMaterial implements Material {
    private final Color baseColor;
    private final Color patternColor;
    private Matrix4 objectTransform;

    // Phong parameters optimized for carpet/wool surface
    private final double ambientCoeff = 0.5; // Higher ambient for fabric
    private final double diffuseCoeff = 0.7;
    private final double specularCoeff = 0.1; // Wool has very weak specular
    private final double shininess = 10.0; // Very rough surface
    private final Color specularColor = new Color(200, 200, 200); // Soft
    specular
    private final double reflectivity = 0.05;
    private final double ior = 1.2;
    private final double transparency = 0.0;

    public CarpetTextureMaterial() {
        this(new Color(170, 0, 0), new Color(40, 40, 40)); // Turkish red with
        dark pattern
    }

    public CarpetTextureMaterial(Color baseColor, Color patternColor) {
        this.baseColor = baseColor;
        this.patternColor = patternColor;
        this.objectTransform = Matrix4.identity();
    }
}
```

```

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // 1. Get carpet pattern color with pile effect
    Color surfaceColor = calculateCarpetColor(worldPoint, worldNormal);

    // 2. Handle light properties (EXACTLY like Checkerboard)
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    // 3. Simulate carpet pile effect on normal
    Vector3 piledNormal = simulatePileEffect(worldPoint, worldNormal);

    // 4. Calculate Phong components with piled normal
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, piledNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(piledNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);
}

```

```

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateCarpetColor(Point3 worldPoint, Vector3 normal) {
    // Transform to object space for pattern calculation
    Point3 localPoint =
objectTransform.inverse().transformPoint(worldPoint);

    // Calculate UV coordinates
    double u = localPoint.x * 6.0;
    double v = localPoint.z * 6.0;

    // Create traditional kilim pattern
    return createKilimPattern(u, v);
}

private Color createKilimPattern(double u, double v) {
    double tileU = (u % 1 + 1) % 1;
    double tileV = (v % 1 + 1) % 1;

    // Border pattern
    if (tileU < 0.08 || tileU > 0.92 || tileV < 0.08 || tileV > 0.92) {
        return patternColor;
    }

    // Diamond pattern in center
    if (Math.abs(tileU - 0.5) + Math.abs(tileV - 0.5) < 0.25) {
        return patternColor;
    }

    // Additional geometric elements
    if ((Math.abs(tileU - 0.25) < 0.06 && Math.abs(tileV - 0.25) < 0.06) ||
        (Math.abs(tileU - 0.75) < 0.06 && Math.abs(tileV - 0.75) < 0.06) ||
        (Math.abs(tileU - 0.25) < 0.06 && Math.abs(tileV - 0.75) < 0.06) ||
        (Math.abs(tileU - 0.75) < 0.06 && Math.abs(tileV - 0.25) < 0.06)) {
        return patternColor;
    }
}

```

```

// Vertical and horizontal lines
if (Math.abs(tileU - 0.5) < 0.02 || Math.abs(tileV - 0.5) < 0.02) {
    return patternColor;
}

return baseColor;
}

private Vector3 simulatePileEffect(Point3 point, Vector3 originalNormal)
{
    // Simulate carpet pile using noise functions
    double pileU = point.x * 8.0;
    double pileV = point.z * 8.0;

    // Create gentle waves in the pile
    double pileX = Math.sin(pileU * 2.0) * 0.1;
    double pileY = 1.0; // Main pile direction (up)
    double pileZ = Math.cos(pileV * 2.0) * 0.1;

    Vector3 pileEffect = new Vector3(pileX, pileY, pileZ).normalize();
    return originalNormal.add(pileEffect).normalize();
}

```

```

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }

```

```
}
```

```

// =====
// File: /net/lena/murat/material/HamamSaunaMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;

```

```

import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class HamamSaunaMaterial implements Material {
    private final Color marbleColor;
    private final Color woodColor;
    private final Color steamColor;
    private final double steamIntensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.5;
    private final double diffuseCoeff = 0.8;
    private final double specularCoeff = 0.25;
    private final double shininess = 40.0;
    private final double reflectivity = 0.15;
    private final double ior = 1.55;
    private final double transparency = 0.12;

    public HamamSaunaMaterial() {
        this(new Color(0xE6, 0xE6, 0xFA), new Color(0x8B, 0x45, 0x13), new
Color(0xF5, 0xF5, 0xF5, 150), 0.55);
    }

    public HamamSaunaMaterial(Color marbleColor, Color woodColor,
Color steamColor, double steamIntensity) {
        this.marbleColor = marbleColor;
        this.woodColor = woodColor;
        this.steamColor = steamColor;
        this.steamIntensity = Math.max(0, Math.min(1, steamIntensity));
        this.objectTransform = Matrix4.identity();
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectTransform = tm;
    }

    @Override

```

```

public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    Point3 objectPoint =
objectTransform.inverse().transformPoint(worldPoint);

    Color surfaceColor = calculateHamamSaunaPattern(objectPoint,
worldNormal, viewerPos);

    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }

    double NdotL = Math.max(0, worldNormal.dot(props.direction));
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
    double RdotV = Math.max(0, reflectDir.dot(viewDir));
    double specFactor = Math.pow(RdotV, shininess) * props.intensity;
    Color specular = ColorUtil.multiplyColors(new Color(0xFF, 0xFA,
0xF0), props.color, specularCoeff * specFactor);

    return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateHamamSaunaPattern(Point3 point, Vector3
normal, Point3 viewerPos) {
    double x = point.x * 8.0;
    double y = point.y * 8.0;
    double z = point.z * 8.0;
}

```

```

// Marble veining patterns (Hamam - Turkish)
double marbleVein1 = Math.sin(x * 2.5 + Math.cos(y * 1.8) * 3.0);
double marbleVein2 = Math.abs(Math.cos(x * 3.0 + y * 2.2) +
Math.sin(y * 2.5 + z * 1.5));
double marbleVein3 = Math.sin(x * 4.0 + y * 3.0) * Math.cos(y * 2.0 +
z * 1.8);

// Wood grain patterns (Sauna - Norwegian)
double woodGrain1 = Math.sin(x * 1.2 + Math.sin(y * 0.5) * 2.0);
double woodGrain2 = (Math.floor(x * 0.6) + Math.floor(y * 0.6)) % 1.8;
double woodGrain3 = Math.abs(Math.sin(x * 2.8 + y * 1.4 + z * 1.0));

// Steam/mist effect
double steamEffect = Math.sin(x * 0.7 + y * 0.9 + z * 1.2) * Math.cos(y
* 1.1 + z * 0.8);

// Cultural fusion pattern
double marbleWeight = 0.6;
double woodWeight = 0.4;

double materialPattern = (marbleVein1 * 0.3 + marbleVein2 * 0.2 +
marbleVein3 * 0.2) * marbleWeight +
(woodGrain1 * 0.25 + woodGrain2 * 0.15 + woodGrain3 * 0.1) *
woodWeight;

double normalizedPattern = (materialPattern + 1.0) * 0.5;
double steamNormalized = (steamEffect + 1.0) * 0.5;

// View-dependent effects for steam
Vector3 viewDir = viewerPos.subtract(point).normalize();
double viewEffect = Math.pow(Math.abs(viewDir.dot(normal)), 0.6);

// Base material selection
Color baseColor;
if (normalizedPattern < 0.4) {
    // Marble background
    double veinIntensity = normalizedPattern / 0.4;
    baseColor = ColorUtil.blendColors(marbleColor,
    ColorUtil.darkenColor(marbleColor, 0.15), veinIntensity);
}

```

```

} else if (normalizedPattern < 0.7) {
    // Wood grain details
    double woodIntensity = (normalizedPattern - 0.4) / 0.3;
    baseColor = ColorUtil.blendColors(woodColor,
        ColorUtil.lightenColor(woodColor, 0.25), woodIntensity);
} else {
    // Marble-wood transition areas
    double blendIntensity = (normalizedPattern - 0.7) / 0.3;
    baseColor = ColorUtil.blendColors(marbleColor, woodColor,
blendIntensity);
}

// Apply steam effect
if (steamNormalized > (0.7 - steamIntensity * 0.3)) {
    double steamAlpha = (steamNormalized - (0.7 - steamIntensity *
0.3)) / (steamIntensity * 0.3 + 0.3);
    steamAlpha = Math.min(1.0, steamAlpha * 1.5);
    return ColorUtil.blendColors(baseColor, steamColor, steamAlpha *
viewEffect);
}

return baseColor;
}

```

`@Override`

```

public double getReflectivity() {
    return reflectivity;
}

```

`@Override`

```

public double getIndexOfRefraction() {
    return ior;
}

```

`@Override`

```

public double getTransparency() {
    return transparency;
}

```

```
}

// =====
// File: /net/elenamurat/material/CrystalMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

public class CrystalMaterial implements Material {
    private Color baseColor;
    private Color crystalColor;
    private double rayDensity;
    private double raySharpness;
    private double transparency;

    public CrystalMaterial(Color baseColor, Color crystalColor,
        double rayDensity, double raySharpness) {
        this.baseColor = baseColor;
        this.crystalColor = crystalColor;
        this.rayDensity = rayDensity;
        this.raySharpness = raySharpness;
        this.transparency = calculateTransparency(baseColor);
    }

    public CrystalMaterial(Color baseColor, Color crystalColor) {
        this(baseColor, crystalColor, 8.0, 0.7);
    }

    private double calculateTransparency(Color color) {
        int alpha = color.getAlpha();
        return 1.0 - ((double)alpha / 255.0);
    }
}
```

```
@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return 0.3;
}

@Override
public double getIndexOfRefraction() {
    return 1.55;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    // Not needed for this material
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    double pattern = calculateCrystalPattern(point);
    return ColorUtil.blendColors(baseColor, crystalColor, pattern);
}

private double calculateCrystalPattern(Point3 point) {
    double distanceFromCenter = Math.sqrt(
        point.x * point.x + point.y * point.y + point.z * point.z
    );

    double angle = Math.atan2(point.y, point.x);
    int numRays = (int) (rayDensity * 6);
    double rayPattern = Math.sin(angle * numRays);
    double pattern = rayPattern * Math.exp(-distanceFromCenter *
raySharpness);
```

```
        return (pattern + 1.0) / 2.0;
    }

    public Color getBaseColor() {
        return baseColor;
    }

    public void setBaseColor(Color baseColor) {
        this.baseColor = baseColor;
        this.transparency = calculateTransparency(baseColor);
    }

    public Color getCrystalColor() {
        return crystalColor;
    }

    public void setCrystalColor(Color crystalColor) {
        this.crystalColor = crystalColor;
    }

    public double getRayDensity() {
        return rayDensity;
    }

    public void setRayDensity(double rayDensity) {
        this.rayDensity = rayDensity;
    }

    public double getRaySharpness() {
        return raySharpness;
    }

    public void setRaySharpness(double raySharpness) {
        this.raySharpness = raySharpness;
    }

}
```

```
// =====
// File: /net/elenamurat/material/TexturedPhongMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;
import java.awt.image.BufferedImage;

import net.elena.murat.light.*;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;
import net.elena.murat.math.Matrix4;
import net.elena.murat.math.Ray;

/**
 * Represents a Phong material with added texture mapping capabilities.
 * It extends the standard Phong illumination model by applying a
BufferedImage as a texture.
 * This version allows for texture offsetting and scaling (tiling) and uses
 * object's inverse transform for correct UV mapping in local space.
 * This material now fully implements the extended Material interface.
 */

public class TexturedPhongMaterial implements Material {
    // Phong material properties
    private final Color baseDiffuseColor; // This color acts as a base if no
texture, or for tinting
    private final Color specularColor;
    private final double shininess;
    private final double ambientCoefficient;
    private final double diffuseCoefficient;
    private final double specularCoefficient;
    private final double reflectivity;
    private final double ior; // Index of Refraction
    private final double transparency;

    // Texture property - now holding BufferedImage directly
    private final BufferedImage texture; // The image to be mapped onto the
surface
```

```

private final double uOffset;           // Horizontal texture offset
private final double vOffset;           // Vertical texture offset
private final double uScale;           // Horizontal texture tiling/scaling
factor
private final double vScale;           // Vertical texture tiling/scaling factor

// The inverse transformation matrix of the object this material is applied
to
private Matrix4 objectInverseTransform;

/***
 * Full constructor for TexturedPhongMaterial.
 * @param baseDiffuseColor The base diffuse color (used as a fallback
or tint for texture).
 * @param specularColor The specular color.
 * @param shininess The shininess exponent for specular highlights.
 * @param ambientCoefficient The ambient light coefficient.
 * @param diffuseCoefficient The diffuse light coefficient.
 * @param specularCoefficient The specular light coefficient.
 * @param reflectivity The reflectivity of the material (0.0 to 1.0).
 * @param ior The index of refraction (for transparent materials).
 * @param transparency The transparency of the material (0.0 to 1.0).
 * @param texture The BufferedImage to be used as a texture. Can be
null.
 * @param uOffset Horizontal offset for the texture (e.g., 0.5 to shift by
half width).
 * @param vOffset Vertical offset for the texture (e.g., 0.5 to shift by half
height).
 * @param uScale Horizontal tiling/scaling factor (e.g., 2.0 to repeat
texture twice).
 * @param vScale Vertical tiling/scaling factor (e.g., 2.0 to repeat texture
twice).
 * @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.
*/
public TexturedPhongMaterial(Color baseDiffuseColor, Color
specularColor, double shininess,
double ambientCoefficient, double diffuseCoefficient, double
specularCoefficient,

```

```

        double reflectivity, double ior, double transparency,
        BufferedImage texture,
        double uOffset, double vOffset, double uScale, double vScale,
        Matrix4 objectInverseTransform) { // Added objectInverseTransform
    this.baseDiffuseColor = baseDiffuseColor;
    this.specularColor = specularColor;
    this.shininess = shininess;
    this.ambientCoefficient = ambientCoefficient;
    this.diffuseCoefficient = diffuseCoefficient;
    this.specularCoefficient = specularCoefficient;
    this.reflectivity = clamp01(reflectivity);
    this.ior = Math.max(1.0, ior); // IOR should be at least 1.0 (for
vacuum/air)
    this.transparency = clamp01(transparency);
    this.texture = texture;
    this.uOffset = uOffset;
    this.vOffset = vOffset;
    this.uScale = uScale;
    this.vScale = vScale;
    this.objectInverseTransform = objectInverseTransform; // Store the
inverse transform
}

```

/**

* Constructor for a textured material with default Phong properties and
no texture transformations.

* @param baseDiffuseColor The base diffuse color.
* @param texture The BufferedImage to be used as a texture.
* @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.

*/

```

public TexturedPhongMaterial(Color baseDiffuseColor, BufferedImage
texture, Matrix4 objectInverseTransform) { // Added
objectInverseTransform
    // Call the full constructor with default values
    this(baseDiffuseColor, Color.WHITE, 32.0, 0.1, 0.7, 0.7, 0.0, 1.0, 0.0,
texture, 0.0, 0.0, 1.0, 1.0, objectInverseTransform);
}

```

```
/***
 * Constructor for a textured material with full Phong properties but no
texture transformations.
 * @param baseDiffuseColor The base diffuse color.
 * @param specularColor The specular color.
 * @param shininess The shininess exponent for specular highlights.
 * @param ambientCoefficient The ambient light coefficient.
 * @param diffuseCoefficient The diffuse light coefficient.
 * @param specularCoefficient The specular light coefficient.
 * @param reflectivity The reflectivity of the material (0.0 to 1.0).
 * @param ior The index of refraction (for transparent materials).
 * @param transparency The transparency of the material (0.0 to 1.0).
 * @param texture The BufferedImage to be used as a texture.
 * @param objectInverseTransform The full inverse transformation
matrix of the object.
 */
```

```
public TexturedPhongMaterial(Color baseDiffuseColor, Color
specularColor, double shininess,
double ambientCoefficient, double diffuseCoefficient, double
specularCoefficient,
double reflectivity, double ior, double transparency,
BufferedImage texture, Matrix4 objectInverseTransform) { // Added
objectInverseTransform
// Call the full constructor with default transformation values
this(baseDiffuseColor, specularColor, shininess, ambientCoefficient,
diffuseCoefficient,
specularCoefficient, reflectivity, ior, transparency, texture, 0.0, 0.0, 1.0,
1.0, objectInverseTransform);
}
```

```
/***
 * Constructor for a Phong material without texture (similar to original
PhongMaterial).
 * Defaults to no texture and default texture transformations.
 * @param diffuseColor The diffuse color of the material.
 * @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.
 */
```

```
public TexturedPhongMaterial(Color diffuseColor, Matrix4
```

```
objectInverseTransform) { // Added objectInverseTransform
    // Call the full constructor with default values (texture as null)
    this(diffuseColor, Color.WHITE, 32.0, 0.1, 0.7, 0.7, 0.0, 1.0, 0.0, null,
0.0, 0.0, 1.0, 1.0, objectInverseTransform);
}

// --- Getters for material properties ---
public Color getBaseDiffuseColor() { return baseDiffuseColor; }
public Color getSpecularColor() { return specularColor; }
public double getShininess() { return shininess; } // Not part of Material
interface, but useful internally
public double getAmbientCoefficient() { return ambientCoefficient; } // // Not part of Material interface
public double getDiffuseCoefficient() { return diffuseCoefficient; } // Not part of Material interface
public double getSpecularCoefficient() { return specularCoefficient; } // // Not part of Material interface
```

@Override

```
public double getReflectivity() { return reflectivity; }
```

@Override

```
public double getIndexOfRefraction() { return ior; }
```

@Override

```
public double getTransparency() { return transparency; }
```

```
public BufferedImage getTexture() { return texture; } // Returns
BufferedImage
```

```
public double getUOffset() { return uOffset; }
public double getVOffset() { return vOffset; }
public double getUScale() { return uScale; }
public double getVScale() { return vScale; }
```

@Override

```
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}
```

```

/**
 * Calculates the final color at a given point on the surface, considering
light sources
 * and Phong illumination model with texture mapping.
 * This is the primary method used by the RayTracer's shading function.
 * @param worldPoint The intersection point on the surface in world
coordinates.
 * @param worldNormal The normal vector at the intersection point in
world coordinates (must be normalized).
 * @param light The light source currently being evaluated.
 * @param viewerPos The position of the viewer/camera.
 * @return The calculated Color at the point.
*/
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // Check if inverse transform is valid before proceeding
    if (objectInverseTransform == null) {
        System.err.println("Error: TexturedPhongMaterial's inverse transform
is null. Returning black.");
        return Color.BLACK;
    }

    // 1. Transform world point and normal to object's local space for UV
mapping
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    // Check if the transformed normal is valid
    if (localNormal == null) {
        System.err.println("Error: TexturedPhongMaterial's local normal is
null. Returning base diffuse color.");
        return baseDiffuseColor; // Fallback to base color if normal
transformation fails
    }
}

```

```

// Get the color from the texture at the intersection point using planar
UV mapping.
// If no texture is present or mapping fails, this will return the
baseDiffuseColor.

Color textureColor = getTextureColor(localPoint, localNormal);
double texR = textureColor.getRed() / 255.0;
double texG = textureColor.getGreen() / 255.0;
double texB = textureColor.getBlue() / 255.0;

double rCombined = 0.0;
double gCombined = 0.0;
double bCombined = 0.0;

double specR = specularColor.getRed() / 255.0;
double specG = specularColor.getGreen() / 255.0;
double specB = specularColor.getBlue() / 255.0;

// Ambient Light calculation
if (light instanceof ElenaMuratAmbientLight) {
    double ambientIntensity = light.getIntensity();
    // Ambient component is scaled by the texture color
    rCombined = texR * ambientCoefficient * ambientIntensity *
(light.getColor().getRed() / 255.0);
    gCombined = texG * ambientCoefficient * ambientIntensity *
(light.getColor().getGreen() / 255.0);
    bCombined = texB * ambientCoefficient * ambientIntensity *
(light.getColor().getBlue() / 255.0);
} else { // Directional, Point, Pulsating, and Spot lights (non-ambient)
    Vector3 lightDir = getLightDirection(light, worldPoint); // Use
worldPoint for light direction
    if (lightDir == null) { // Handle unsupported light types
        return Color.BLACK;
    }
    Color lightColorFromSource = light.getColor(); // Get color from the
light source
    double attenuatedIntensity;

// Get attenuated intensity based on light type

```

```

if (light instanceof MuratPointLight) {
    attenuatedIntensity = ((MuratPointLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof ElenaDirectionalLight) {
    attenuatedIntensity = ((ElenaDirectionalLight) light).getIntensity();
} else if (light instanceof PulsatingPointLight) {
    attenuatedIntensity = ((PulsatingPointLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof SpotLight) {
    attenuatedIntensity = ((SpotLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof BioluminescentLight) {
    attenuatedIntensity = ((BioluminescentLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof BlackHoleLight) {
    attenuatedIntensity = ((BlackHoleLight)
light).getAttenuatedIntensity(worldPoint);
} else if (light instanceof FractalLight) {
    attenuatedIntensity = ((FractalLight)
light).getAttenuatedIntensity(worldPoint);
} else {
    // Bu else bloğuna düşmemesi gerekiyor, çünkü getLightDirection
zaten kontrol ediyor.
    // Ancak bir güvenlik önlemi olarak burada da bir uyarı basabiliriz.
    System.err.println("Warning: Unknown or unsupported light type for
TexturedPhongMaterial shading (intensity): " +
light.getClass().getName());
    return Color.BLACK;
}

// Diffuse component: scaled by the texture color
double diffuseFactor = Math.max(0, worldNormal.dot(lightDir)); //
Use worldNormal for lighting
rCombined = texR * diffuseCoefficient * diffuseFactor *
(lightColorFromSource.getRed() / 255.0) * attenuatedIntensity;
gCombined = texG * diffuseCoefficient * diffuseFactor *
(lightColorFromSource.getGreen() / 255.0) * attenuatedIntensity;
bCombined = texB * diffuseCoefficient * diffuseFactor *
(lightColorFromSource.getBlue() / 255.0) * attenuatedIntensity;

```

```

// Specular component: not usually scaled by texture, uses material's
specularColor
    Vector3 V = viewerPos.subtract(worldPoint).normalize();
    Vector3 R = lightDir.negate().reflect(worldNormal); // Reflect the light
direction vector (negated for incoming light)

    double specularFactor = Math.max(0, R.dot(V));
    specularFactor = Math.pow(specularFactor, shininess);

    rCombined += specR * specularCoefficient * specularFactor *
(lightColorFromSource.getRed() / 255.0) * attenuatedIntensity;
    gCombined += specG * specularCoefficient * specularFactor *
(lightColorFromSource.getGreen() / 255.0) * attenuatedIntensity;
    bCombined += specB * specularCoefficient * specularFactor *
(lightColorFromSource.getBlue() / 255.0) * attenuatedIntensity;
}

// Clamp final color values to [0, 1] range and convert to Color object
return new Color((float)clamp01(rCombined),
(float)clamp01(gCombined), (float)clamp01(bCombined));
}

/**
 * Helper method to clamp a double value between 0.0 and 1.0.
 * @param val The value to clamp.
 * @return The clamped value.
 */
private double clamp01(double val) {
    return Math.min(1.0, Math.max(0.0, val));
}

/**
 * Retrieves the color from the texture image at the given local
intersection point,
 * using planar UV mapping based on the provided local normal vector,
and applies
 * offset and scale transformations.
 * @param localPoint The intersection point in the object's local

```

coordinates.

* @param localNormal The normalized normal vector at the intersection point in local coordinates.

* @return The Color sampled from the texture, or the baseDiffuseColor if no texture is loaded.

*/

```
private Color getTextureColor(Point3 localPoint, Vector3 localNormal) {
```

```
    if (texture == null) {
```

```
        return baseDiffuseColor; // If no texture is loaded, return the base  
        diffuse color
```

```
}
```

```
    if (localNormal == null) {
```

```
        System.err.println("Warning: getTextureColor called with a null local  
        normal. Cannot perform planar UV mapping. Returning base diffuse  
        color.");
```

```
        return baseDiffuseColor;
```

```
}
```

```
// Planar UV mapping based on the dominant axis of the local normal.
```

```
// This is consistent with other textured materials (Squared,  
Checkerboard, Striped, Circle).
```

```
double u, v;
```

```
double absNx = Math.abs(localNormal.x);
```

```
double absNy = Math.abs(localNormal.y);
```

```
double absNz = Math.abs(localNormal.z);
```

```
// Project the 3D local point onto a 2D plane based on the dominant  
local normal axis.
```

```
// Add a small epsilon to the coordinates before flooring to handle  
floating point inaccuracies at boundaries.
```

```
if (absNx > absNy && absNx > absNz) { // Normal is mostly X-axis  
(local Y-Z plane)
```

```
    u = localPoint.y + Ray.EPSILON;
```

```
    v = localPoint.z + Ray.EPSILON;
```

```
} else if (absNy > absNx && absNy > absNz) { // Normal is mostly Y-  
axis (local X-Z plane)
```

```
    u = localPoint.x + Ray.EPSILON;
```

```
    v = localPoint.z + Ray.EPSILON;
```

```
} else { // Normal is mostly Z-axis (local X-Y plane) or equally
```

dominant

```
    u = localPoint.x + Ray.EPSILON;  
    v = localPoint.y + Ray.EPSILON;  
}
```

```
// Apply scaling (tiling) and offset (shifting)  
u = (u * uScale) + uOffset;  
v = (v * vScale) + vOffset;
```

```
// Wrap UV coordinates to [0,1) range for seamless tiling  
u = (u % 1.0 + 1.0) % 1.0;  
v = (v % 1.0 + 1.0) % 1.0;
```

```
// Map UV coordinates to texture pixel coordinates  
int texX = (int) (u * (texture.getWidth() - 1));  
int texY = (int) (v * (texture.getHeight() - 1));
```

```
// Clamp texture coordinates to ensure they are within bounds [0, width-1] and [0, height-1]  
texX = Math.min(Math.max(0, texX), texture.getWidth() - 1);  
texY = Math.min(Math.max(0, texY), texture.getHeight() - 1);
```

```
// Get the RGB integer value from the texture
```

```
int rgb = texture.getRGB(texX, texY);
```

```
// Ensure alpha channel is fully opaque (255) to prevent unexpected transparency issues
```

```
return new Color(rgb | 0xFF000000);
```

```
}
```

```
/**
```

```
 * Helper method to get the normalized light direction vector from a light source to a point.
```

```
 * @param light The light source.
```

```
 * @param worldPoint The point in world coordinates.
```

```
 * @return Normalized light direction vector, or null if light type is unsupported.
```

```
*/
```

```
private Vector3 getLightDirection(Light light, Point3 worldPoint) {  
    if (light instanceof MuratPointLight) {
```

```

        return
((MuratPointLight)light).getPosition().subtract(worldPoint).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return
((ElenaDirectionalLight)light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return
((PulsatingPointLight)light).getPosition().subtract(worldPoint).normalize()
;
    } else if (light instanceof SpotLight) {
        return ((SpotLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof BioluminescentLight) {
        return
((BioluminescentLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof BlackHoleLight) {
        return
((BlackHoleLight)light).getDirectionAt(worldPoint).normalize();
    } else if (light instanceof FractalLight) {
        return ((FractalLight)light).getDirectionAt(worldPoint).normalize();
    } else {
        System.err.println("Warning: Unknown light type in
PlaneFaceElenaTextureMaterial");
        return new Vector3(0, 1, 0); // Varsayılan yön
    }
}
}

```

```

// =====
// File: /net/elenamurat/material/VikingMetalMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

```

```
import net.elena.murat.util.ColorUtil;

public class VikingMetalMaterial implements Material {
    private final Color baseColor;
    private final Color rustColor;
    private final double rustDensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.4;
    private final double diffuseCoeff = 0.7;
    private final double specularCoeff = 0.15;
    private final double shininess = 15.0;
    private final double reflectivity = 0.08;
    private final double ior = 2.8;
    private final double transparency = 0.0;

    public VikingMetalMaterial() {
        this(new Color(0x22, 0x22, 0x22), new Color(0xBB, 0x55, 0x22),
            0.35);
    }

    public VikingMetalMaterial(Color baseColor, Color rustColor, double
        rustDensity) {
        this.baseColor = baseColor;
        this.rustColor = rustColor;
        this.rustDensity = Math.max(0, Math.min(1, rustDensity));
        this.objectTransform = Matrix4.identity();
    }

    @Override
    public void setObjectTransform(Matrix4 tm) {
        if (tm == null) tm = new Matrix4 ();
        this.objectTransform = tm;
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
        light, Point3 viewerPos) {
        Point3 objectPoint =
```

```

objectTransform.inverse().transformPoint(worldPoint);

Color surfaceColor = calculateRustPattern(objectPoint);

LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
if (props == null) return surfaceColor;

Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(Color.LIGHT_GRAY,
props.color, specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateRustPattern(Point3 point) {
    double noise = simplexNoise3D(point.x * 5, point.y * 5, point.z * 5);

    double t = (noise + 1) * 0.5;

    if (t < rustDensity) {
        return interpolateColor(rustColor,
            new Color(Math.min(255, (int)(rustColor.getRed() * 0.8)),
            Math.min(255, (int)(rustColor.getGreen() * 0.9)),
            Math.min(255, (int)(rustColor.getBlue() * 1.1))));
    }
}

```

```

t / rustDensity);
} else {
    return interpolateColor(baseColor,
        new Color(Math.min(255, (int)(baseColor.getRed() * 1.2)),
            Math.min(255, (int)(baseColor.getGreen() * 1.1)),
            Math.min(255, (int)(baseColor.getBlue() * 0.9))),
        (t - rustDensity) / (1 - rustDensity));
}
}

private double simplexNoise3D(double x, double y, double z) {
    double value = Math.sin(x * 0.431) + Math.sin(y * 0.723) + Math.sin(z
* 0.327);
    value += Math.sin(x * 1.531 + y * 0.927) * 0.5;
    value += Math.sin(y * 1.231 + z * 1.627) * 0.25;
    return value % 1.0;
}

private Color interpolateColor(Color c1, Color c2, double t) {
    t = Math.max(0, Math.min(1, t));
    int r = (int)(c1.getRed() * (1-t) + c2.getRed() * t);
    int g = (int)(c1.getGreen() * (1-t) + c2.getGreen() * t);
    int b = (int)(c1.getBlue() * (1-t) + c2.getBlue() * t);
    return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return ior;
}

@Override
public double getTransparency() {
    return transparency;
}

```

```
}

}

// =====
// File: /net/elenamurat/material/ThresholdMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;

public class ThresholdMaterial implements Material {

    private Color baseColor;
    private double threshold;
    private Color aboveColor;
    private Color belowColor;
    private boolean useLightColor;
    private boolean invertThreshold;

    private double transparency = 0.0; //opaque

    public ThresholdMaterial(Color baseColor, double threshold) {
        this(baseColor, threshold, Color.WHITE, Color.BLACK, false, false);
    }

    public ThresholdMaterial(Color baseColor, double threshold, Color
aboveColor, Color belowColor) {
        this(baseColor, threshold, aboveColor, belowColor, false, false);
    }

    public ThresholdMaterial(Color baseColor, double threshold, Color
aboveColor,
        Color belowColor, boolean useLightColor, boolean invertThreshold) {
```

```

this.baseColor = baseColor;
this.threshold = Math.max(0.0, Math.min(1.0, threshold));
this.aboveColor = aboveColor;
this.belowColor = belowColor;
this.useLightColor = useLightColor;
this.invertThreshold = invertThreshold;
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Color sourceColor = baseColor;

    if (useLightColor) {
        sourceColor = light.getColor();
    }

    int alfa = sourceColor.getAlpha ();
    double alpha = ((double)(alfa))/255.0;

    setTransparency (1-alpha);

    return applyThreshold(sourceColor, threshold, aboveColor, belowColor,
invertThreshold);
}

private Color applyThreshold(Color color, double thresholdValue, Color
above, Color below, boolean invert) {
    float[] rgb = color.getRGBColorComponents(null);

    double intensity = 0.299 * rgb[0] + 0.587 * rgb[1] + 0.114 * rgb[2];

    if (invert) {
        intensity = 1.0 - intensity;
    }

    return intensity > thresholdValue ? above : below;
}

```

```
@Override
public double getReflectivity() {
    return 0.0;
}

@Override
public double getTransparency() {
    return transparency;
}

public void setTransparency (double tnw){
    this.transparency = tnw;
}

@Override
public double getIndexOfRefraction() {
    return 1.0;
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

// Getters Setters
public Color getBaseColor() {
    return baseColor;
}

public void setBaseColor(Color baseColor) {
    this.baseColor = baseColor;
}

public double getThreshold() {
    return threshold;
}

public void setThreshold(double threshold) {
    this.threshold = Math.max(0.0, Math.min(1.0, threshold));
}
```

```
public Color getAboveColor() {
    return aboveColor;
}

public void setAboveColor(Color aboveColor) {
    this.aboveColor = aboveColor;
}

public Color getBelowColor() {
    return belowColor;
}

public void setBelowColor(Color belowColor) {
    this.belowColor = belowColor;
}

public boolean isUseLightColor() {
    return useLightColor;
}

public void setUseLightColor(boolean useLightColor) {
    this.useLightColor = useLightColor;
}

public boolean isInvertThreshold() {
    return invertThreshold;
}

public void setInvertThreshold(boolean invertThreshold) {
    this.invertThreshold = invertThreshold;
}

public static double calculateAutoThreshold(Color color) {
    float[] rgb = color.getRGBColorComponents(null);
    double intensity = 0.299 * rgb[0] + 0.587 * rgb[1] + 0.114 * rgb[2];
    return intensity;
}
```

```
}
```

```
// =====  
// File: /net/elenamurat/material/ImageTexture.java  
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;  
import java.awt.image.BufferedImage;
```

```
import net.elena.murat.math.*;
```

```
/**
```

```
 * Represents an image-based texture. It can return a color based on  
 * 3D point and normal (using spherical mapping for spheres) or direct UV  
 coordinates.
```

```
 * This class does NOT implement the Material interface directly, but is  
 used by Materials.
```

```
 */
```

```
public class ImageTexture {  
    private final BufferedImage image;  
    private final double scaleU;  
    private final double scaleV;  
    private final double offsetU;  
    private final double offsetV;
```

```
    public ImageTexture(BufferedImage image, double scale) {  
        this(image, scale, scale, 0.0, 0.0);  
    }
```

```
    public ImageTexture(BufferedImage image, double scaleU, double  
scaleV, double offsetU, double offsetV) {  
        this.image = image;  
        this.scaleU = scaleU;  
        this.scaleV = scaleV;  
        this.offsetU = offsetU;  
        this.offsetV = offsetV;
```

```

}

/***
 * Retrieves the color from the texture at the given UV coordinates.
 * This method handles texture tiling based on the scale factors.
 * @param u The U-coordinate (horizontal, expected 0.0 to 1.0, but can
be outside for tiling).
 * @param v The V-coordinate (vertical, expected 0.0 to 1.0, but can be
outside for tiling).
 * @return The Color at the specified texture coordinates.
*/
public Color getColorFromUV(double u, double v) { // Renamed to avoid
conflict with Material.getColorAt
    // Apply scaling and offset
    u = u * scaleU + offsetU;
    v = v * scaleV + offsetV;

    // Apply tiling using Math.floorMod for correct wrapping for negative
values
    int imgX = Math.floorMod((int)(u * image.getWidth()),
image.getWidth());
    int imgY = Math.floorMod((int)(v * image.getHeight()),
image.getHeight());

    // Ensure indices are within bounds (should be handled by floorMod, but
as a safeguard)
    imgX = Math.max(0, Math.min(image.getWidth() - 1, imgX));
    imgY = Math.max(0, Math.min(image.getHeight() - 1, imgY));

    return new Color(image.getRGB(imgX, imgY));
}

/***
 * Calculates UV coordinates from a 3D point and its normal using
spherical mapping,
 * then retrieves the color from the texture.
 * This method is generally suitable for spheres centered at the origin.
 * For other shapes, this mapping might not be appropriate.
 *

```

```

* @param point The 3D intersection point on the surface.
* @param normal The surface normal at the intersection point.
* @return The Color from the texture at the calculated UV coordinates.
*/
public Color getColorFrom3DPoint(Point3 point, Vector3 normal) { // Renamed to clarify its purpose
    // Normalize normal to avoid issues with length
    Vector3 n = normal.normalize();

    // Spherical mapping from normal (latitude/longitude)
    // Theta: angle from +Y axis (pole), range [0, PI]
    double theta = Math.acos(Math.max(-1.0, Math.min(1.0, n.y))); // Clamp n.y for numerical stability

    // Phi: angle around Y axis, range [-PI, PI] then adjusted to [0, 2PI]
    double phi = Math.atan2(n.z, n.x);
    if (phi < 0) phi += 2 * Math.PI;

    // Convert to UV coordinates (normalized 0.0 to 1.0)
    double u = phi / (2 * Math.PI);
    double v = theta / Math.PI;

    // Get color using the more general getColorFromUV method
    return getColorFromUV(u, v);
}

}

```

```

// =====
// File: /net/elenamurat/material/XRayMaterial.java
// =====

```

```

package net.elenamurat.material;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

```

```
import net.elena.murat.math.*;
import net.elena.murat.light.Light;

public class XRayMaterial implements Material {
    private Matrix4 objectTransform;
    private Matrix4 inverseObjectTransform;

    private final Color baseColor;
    private final double transparency;
    private final double reflectivity;

    private final float RF;
    private final float GF;
    private final float BF;
    private final float AF;

    public XRayMaterial() {
        this.baseColor = new Color (0.15f, 0.6f, 1.0f, 0.6f);
        this.transparency = 0.92;
        this.reflectivity = 0.05;

        this.RF = ((float)(this.baseColor.getRed ())/(255.0f));
        this.GF = ((float)(this.baseColor.getGreen ())/(255.0f));
        this.BF = ((float)(this.baseColor.getBlue ())/(255.0f));
        this.AF = ((float)(this.baseColor.getAlpha ())/(255.0f));

        this.objectTransform = Matrix4.identity();
        this.inverseObjectTransform = Matrix4.identity();
    }

    public XRayMaterial(Color baseColor, double transparency, double
reflectivity) {
        this.baseColor = baseColor;
        this.transparency = transparency;
        this.reflectivity = reflectivity;

        this.RF = ((float)(this.baseColor.getRed ())/(255.0f));
        this.GF = ((float)(this.baseColor.getGreen ())/(255.0f));
        this.BF = ((float)(this.baseColor.getBlue ())/(255.0f));
```

```
this.AF = ((float)(this.baseColor.getAlpha ()))/(255.0f);

this.objectTransform = Matrix4.identity();
this.inverseObjectTransform = Matrix4.identity();
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {
    Point3 localPoint = inverseObjectTransform.transformPoint(point);

    double depth = calculateDepth(localPoint);
    float intensity = (float) Math.exp(-depth * 0.5);

    return new Color(
        RF,
        GF * intensity,
        BF * intensity,
        AF
    );
}

private double calculateDepth(Point3 point) {
    return point.length();
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return 1.1;
```

```
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4();
    this.objectTransform = tm;
    this.inverseObjectTransform = tm.inverse();
}

// =====
// File: /net/elenamurat/material/CheckerboardMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class CheckerboardMaterial implements Material {
    private final Color color1;
    private final Color color2;
    private final double size;
    private Matrix4 objectInverseTransform;

    // Phong parameters
    private final double ambientCoeff;
    private final double diffuseCoeff;
    private final double specularCoeff;
    private final double shininess;
    private final Color specularColor;
    private final double reflectivity;
    private final double ior;
    private final double transparency;
```

```
public CheckerboardMaterial(Color color1, Color color2, double size,  
Matrix4 objectInverseTransform) {  
    this(color1, color2, size, 0.1, 0.7, 0.8, 50.0, Color.WHITE, 0.0, 1.0, 0.0,  
    objectInverseTransform);  
}
```

```
public CheckerboardMaterial(Color color1, Color color2, double size,  
    double ambientCoeff, double diffuseCoeff,  
    double specularCoeff, double shininess, Color specularColor,  
    double reflectivity, double ior, double transparency,  
    Matrix4 objectInverseTransform) {  
    this.color1 = color1;  
    this.color2 = color2;  
    this.size = size;  
    this.objectInverseTransform = objectInverseTransform;  
    this.ambientCoeff = ambientCoeff;  
    this.diffuseCoeff = diffuseCoeff;  
    this.specularCoeff = specularCoeff;  
    this.shininess = shininess;  
    this.specularColor = specularColor;  
    this.reflectivity = reflectivity;  
    this.ior = ior;  
    this.transparency = transparency;  
}
```

```
@Override  
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4 ();  
    this.objectInverseTransform = tm;  
}
```

```
@Override  
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light  
light, Point3 viewerPos) {  
    // 1. Get checkerboard pattern color  
    Point3 localPoint =  
    objectInverseTransform.transformPoint(worldPoint);  
    Color patternColor = calculatePatternColor(localPoint, worldNormal);
```

```

// 2. Handle light properties
LightProperties props = getLightPropertiesX(light, worldPoint);
if (props == null) return patternColor;

// 3. Calculate Phong components
Color ambient = ColorUtil.multiplyColors(patternColor, props.color,
ambientCoeff);

if (light instanceof ElenaMuratAmbientLight) {
    return ambient;
}

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(patternColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculatePatternColor(Point3 localPoint, Vector3
worldNormal) {
    Vector3 localNormal =
objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    double u, v;
    double absNx = Math.abs(localNormal.x);
    double absNy = Math.abs(localNormal.y);
    double absNz = Math.abs(localNormal.z);

    if (absNx > absNy && absNx > absNz) {

```

```

u = localPoint.y * size + Ray.EPSILON;
v = localPoint.z * size + Ray.EPSILON;
} else if (absNy > absNx && absNy > absNz) {
u = localPoint.x * size + Ray.EPSILON;
v = localPoint.z * size + Ray.EPSILON;
} else {
u = localPoint.x * size + Ray.EPSILON;
v = localPoint.y * size + Ray.EPSILON;
}
}

return ((int)Math.floor(u) + (int)Math.floor(v)) % 2 == 0 ? color1 :
color2;
}

```

```
private LightProperties getLightPropertiesX(Light light, Point3 point) {
if (light == null) return null;
```

```
if (light instanceof ElenaMuratAmbientLight) {
return new LightProperties(
new Vector3(0, 0, 0), // Dummy direction
light.getColor(),
light.getIntensity()
);
}
```

```
Vector3 dir;
double intensity;
```

```
try {
// Handle all light types consistently
if (light instanceof MuratPointLight ||
light instanceof PulsatingPointLight ||
light instanceof BioluminescentLight) {
dir = light.getPosition().subtract(point).normalize();
intensity = light.getAttenuatedIntensity(point);
}
else if (light instanceof ElenaDirectionalLight) {
dir =
((ElenaDirectionalLight)light).getDirection().negate().normalize();
```

```

        intensity = light.getIntensity();
    }
    else if (light instanceof SpotLight ||
        light instanceof BlackHoleLight ||
        light instanceof FractalLight) {
        // Use getDirectionAt() for advanced lights
        dir = light.getDirectionAt(point);
        intensity = light.getAttenuatedIntensity(point);
    }
    else {
        // Fallback for unknown lights
        dir = new Vector3(0, 1, 0); // Default upward direction
        intensity = light.getIntensity();
    }

    return new LightProperties(dir, light.getColor(), intensity);
} catch (Exception e) {
    // Fallback if any calculation fails
    return new LightProperties(
        new Vector3(0, 1, 0),
        light.getColor(),
        Math.min(light.getIntensity(), 1.0)
    );
}
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }

}

// =====
// File: /net/elenamurat/material/DiagonalCheckerMaterial.java
// =====

package net.elena.murat.material;

```

```

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;

/**
 * DiagonalCheckerMaterial represents a surface with a two-color diagonal
 * checkerboard pattern.
 * It fully implements the Material interface, including properties for
 * reflectivity,
 * index of refraction, and transparency.
 */
public class DiagonalCheckerMaterial implements Material {
    private final Color color1; // First color for the checkerboard pattern
    private final Color color2; // Second color for the checkerboard pattern
    private final double scale; // Represents the frequency of squares (squares
    per unit)
    private Matrix4 objectInverseTransform; // The inverse transformation
    matrix of the object

    // Phong lighting model parameters
    private final double ambient; // Ambient reflection coefficient (0.0-1.0)
    private final double diffuse; // Diffuse reflection coefficient (0.0-1.0)
    private final double specular; // Specular reflection coefficient (0.0-1.0)
    private final double shininess; // Shininess exponent for specular
    highlights
    private final Color specularColor; // Color of the specular highlight
    private final double reflectivity; // Reflectivity coefficient (0.0-1.0)
    private final double ior; // Index of Refraction for transparent materials
    private final double transparency; // Transparency coefficient (0.0-1.0)

    /**
     * Constructs a DiagonalCheckerMaterial with two colors, a scale, Phong
     lighting model parameters,
     * and the object's inverse transformation matrix.
     *
     * @param color1          The first color for the checkerboard pattern.
     * @param color2          The second color for the checkerboard pattern.
     * @param scale           The frequency of the squares (e.g., 4.0 for 4

```

squares per unit length).

* @param ambient	The ambient reflection coefficient (0.0-1.0).
* @param diffuse	The diffuse reflection coefficient (0.0-1.0).
* @param specular	The specular reflection coefficient (0.0-1.0).
* @param shininess	The shininess exponent for specular highlights.
* @param specularColor	The color of the specular highlight.
* @param reflectivity	The reflectivity coefficient (0.0-1.0).
* @param ior	The Index of Refraction for transparent materials.
* @param transparency	The transparency coefficient (0.0-1.0).
* @param objectInverseTransform	The full inverse transformation matrix of the object this material is applied to.

*/

```
public DiagonalCheckerMaterial(Color color1, Color color2, double scale,
```

```
    double ambient, double diffuse, double specular,
    double shininess, Color specularColor,
    double reflectivity, double ior, double transparency,
    Matrix4 objectInverseTransform) {
    this.color1 = color1;
    this.color2 = color2;
    this.scale = scale;
    this.objectInverseTransform = objectInverseTransform;
    this.ambient = ambient;
    this.diffuse = diffuse;
    this.specular = specular;
    this.shininess = shininess;
    this.specularColor = specularColor;
    this.reflectivity = reflectivity;
    this.ior = ior;
    this.transparency = transparency;
}
```

```
/**
```

```
 * Simplified constructor for basic diagonal checkerboard pattern without full Phong parameters.
```

```
 * Uses default Phong values.
```

```
*
```

```
* @param color1      The first color for the checkerboard pattern.  
* @param color2      The second color for the checkerboard pattern.  
* @param scale        The frequency of the squares.  
* @param objectInverseTransform The full inverse transformation  
matrix of the object.
```

```
*/
```

```
public DiagonalCheckerMaterial(Color color1, Color color2, double  
scale,  
Matrix4 objectInverseTransform) {  
    this(color1, color2, scale,  
        0.1, 0.7, 0.8, 50.0, Color.WHITE,  
        0.0, 1.0, 0.0, objectInverseTransform);  
}
```

```
@Override
```

```
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4 ();  
    this.objectInverseTransform = tm;  
}
```

```
/**
```

```
* Calculates the color contribution of a single light source at a given  
point on the surface,
```

```
* taking into account surface properties and a diagonal checkerboard  
pattern using the Phong model.
```

```
* The hit point is transformed into the material's local space before  
pattern calculation.
```

```
*
```

```
* @param worldPoint    The point in 3D space (world coordinates)  
where the light hits.
```

```
* @param worldNormal   The normal vector at the point (world  
coordinates).
```

```
* @param light          The single light source affecting this point.
```

```
* @param viewerPos      The position of the viewer/camera.
```

```
* @return               The color contribution from this specific light for  
the point.
```

```
*/
```

```
@Override
```

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal,
```

```

Light light, Point3 viewerPos) {
    // Check if inverse transform is valid before proceeding
    if (objectInverseTransform == null) {
        System.err.println("Error: DiagonalCheckerMaterial's inverse
transform is null. Returning black.");
        return Color.BLACK;
    }

    // 1. Transform point to object's local space
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);

    // 2. Transform the world normal to local space to determine the local
face orientation
    // Normals transform with the inverse transpose of the model matrix.
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

    // Check if the transformed normal is valid
    if (localNormal == null) {
        System.err.println("Error: DiagonalCheckerMaterial's normal
transform matrix is null or invalid. Returning black.");
        return Color.BLACK;
    }

    // Determine the dominant axis of the *local normal* to decide 2D
projection for pattern.
    // This ensures the square pattern aligns correctly with the object's local
faces.
    double absNx = Math.abs(localNormal.x);
    double absNy = Math.abs(localNormal.y);
    double absNz = Math.abs(localNormal.z);

    // Project the 3D local point onto a 2D plane based on the dominant
local normal axis.
    // Normalize coordinates from [-0.5, 0.5] to [0, 1] for a unit cube local
space.
    // Then scale by 'this.scale' which represents squares per unit length.

```

```

// Add a small epsilon to the coordinates before flooring to handle
floating point inaccuracies at boundaries.
double u, v; // 2D texture coordinates
if (absNx > absNy && absNx > absNz) { // Normal is mostly X-axis
(local Y-Z plane)
    u = (localPoint.y + 0.5 + Ray.EPSILON) * this.scale;
    v = (localPoint.z + 0.5 + Ray.EPSILON) * this.scale;
} else if (absNy > absNx && absNy > absNz) { // Normal is mostly Y-
axis (local X-Z plane)
    u = (localPoint.x + 0.5 + Ray.EPSILON) * this.scale;
    v = (localPoint.z + 0.5 + Ray.EPSILON) * this.scale;
} else { // Normal is mostly Z-axis (local X-Y plane) or equally
dominant
    u = (localPoint.x + 0.5 + Ray.EPSILON) * this.scale;
    v = (localPoint.y + 0.5 + Ray.EPSILON) * this.scale;
}

// Use diagonal checkerboard logic for all surfaces.
// The parity of the sum of the integer parts determines the color.
int checkU = (int) Math.floor(u);
int checkV = (int) Math.floor(v);
if ((checkU + checkV) % 2 == 0) { // Diagonal checkerboard pattern
    return this.color1;
} else {
    return this.color2;
}
}

/***
* Returns the reflectivity coefficient of the material.
*
* @return The reflectivity value (0.0-1.0).
*/
@Override
public double getReflectivity() {
    return reflectivity;
}

/**

```

```

 * Returns the index of refraction (IOR) of the material.
 *
 * @return The index of refraction.
 */
@Override
public double getIndexOfRefraction() {
    return ior;
}

/***
 * Returns the transparency coefficient of the material.
 *
 * @return The transparency value (0.0-1.0).
 */
@Override
public double getTransparency() {
    return transparency;
}

}

// =====
// File: /net/lena/murat/material/StripeDirection.java
// =====

package net.elena.murat.material;

/**
 * Enum to define the direction of the stripes.
 */
public enum StripeDirection {
    HORIZONTAL, // Stripes vary along the V-axis (or Y-axis in local
    space)
    VERTICAL, // Stripes vary along the U-axis (or X-axis in local space)
    DIAGONAL, // Stripes vary along a diagonal (U+V or U-V in local
    space)
    RANDOM
}

```

```

// =====
// File: /net/elenamurat/material/ProceduralCloudMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.*;
import net.elena.murat.math.*;
import net.elena.murat.util.ColorUtil;

public class ProceduralCloudMaterial implements Material {
    private final Color baseColor;
    private final Color highlightColor;

    public ProceduralCloudMaterial(Color baseColor, Color highlightColor)
    {
        this.baseColor = baseColor;
        this.highlightColor = highlightColor;
    }

    @Override
    public Color getColorAt(Point3 point, Vector3 normal, Light light,
    Point3 viewerPos) {
        double value = Math.sin(point.x * 5.0) + Math.cos(point.y * 7.0);
        value = (value + 2.0) / 4.0; // normalize 0..1

        int r = (int)(baseColor.getRed() * (1 - value) + highlightColor.getRed()
        * value);
        int g = (int)(baseColor.getGreen() * (1 - value) +
        highlightColor.getGreen() * value);
        int b = (int)(baseColor.getBlue() * (1 - value) +
        highlightColor.getBlue() * value);

        r = ColorUtil.clampColorValue (r);
        g = ColorUtil.clampColorValue (g);
    }
}

```

```
b = ColorUtil.clampColorValue (b);

    return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return 0.0;
}

@Override
public double getIndexOfRefraction() {
    return 1.0;
}

@Override
public double getTransparency() {
    return 0.0; //opaque
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

// =====
// File: /net/elenamurat/material/BumpMaterial.java
// =====

package net.elena.murat.material;

import net.elena.murat.math.Vector3;
import net.elena.murat.math.Point3;
import net.elena.murat.math.Matrix4;
import net.elena.murat.math.Matrix3;
import net.elena.murat.math.Ray;
import net.elena.murat.light.Light;
```

```
import java.awt.Color;
import java.awt.image.BufferedImage;

/***
 * BumpMaterial applies a normal map to perturb the surface normal,
creating the illusion of detail
 * without adding more geometry. It wraps a base material for actual color
calculation.
 * This material now fully implements the extended Material interface.
 */

public class BumpMaterial implements Material {
    private Material baseMaterial; // The underlying material whose lighting
will be perturbed
    private ImageTexture normalMap; // The normal map texture (uses
ImageTexture class)
    private double strength; // The strength of the bump effect (0.0 = no
effect, 1.0 = full effect)
    private Matrix4 objectInverseTransform; // Inverse transform of the
object this material is applied to
    private double uvScale; // Scale factor for UV coordinates when sampling
the normal map

    /**
     * Constructs a BumpMaterial with a base material, a normal map, and a
strength.
     * @param baseMaterial The base material (e.g., PhongMaterial,
LambertMaterial).
     * @param normalMap The ImageTexture object containing the normal
map.
     * @param strength The strength of the bump effect (typically 0.0 to 1.0).
     * @param uvScale The scaling factor for UV coordinates when sampling
the normal map (e.g., 1.0 for 1 repetition per unit).
     * @param objectInverseTransform The full inverse transformation
matrix of the object this material is applied to.
     */
    public BumpMaterial(Material baseMaterial, ImageTexture normalMap,
double strength, double uvScale, Matrix4 objectInverseTransform) {
        this.baseMaterial = baseMaterial;
```

```

this.normalMap = normalMap;
this.strength = strength;
this.uvScale = uvScale;
this.objectInverseTransform = objectInverseTransform;
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

// Getter methods for BumpMaterial's specific properties
public ImageTexture getNormalMap() {
    return normalMap;
}

public double getStrength() {
    return strength;
}

/***
 * Calculates the final shaded color at a given 3D point on the surface,
 * applying the bump mapping effect by perturbing the normal before
 * delegating to the base material for actual color calculation.
 *
 * @param worldPoint The 3D point on the surface in world coordinates.
 * @param worldNormal The geometric surface normal at that point in
 * world coordinates (should be normalized).
 * @param light The light source.
 * @param viewerPos The position of the viewer/camera.
 * @return The final shaded color.
 */
@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // Ensure inverse transform is valid
    if (objectInverseTransform == null) {
        System.err.println("Error: BumpMaterial's inverse transform is null.

```

```

Returning black.");
    return Color.BLACK;
}
// If no normal map, just use the base material with the original normal.
// We check if normalMap is null, as ImageTexture itself might be null.
if (normalMap == null) {
    return baseMaterial.getColorAt(worldPoint, worldNormal, light,
viewerPos);
}

// 1. Transform world point and geometric normal to object's local space
for UV mapping
Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
Vector3 localGeometricNormal =
objectInverseTransform.inverseTransposeForNormal().transformVector(w
orldNormal).normalize();

if (localGeometricNormal == null) {
    System.err.println("Error: BumpMaterial's local geometric normal is
null. Returning base material color with original normal.");
    return baseMaterial.getColorAt(worldPoint, worldNormal, light,
viewerPos);
}

// 2. Calculate UV coordinates based on local point and local geometric
normal
double u, v;
double absNx = Math.abs(localGeometricNormal.x);
double absNy = Math.abs(localGeometricNormal.y);
double absNz = Math.abs(localGeometricNormal.z);

// Planar UV mapping: project onto the dominant plane of the local
normal
// Add a small epsilon to the coordinates before flooring to handle
floating point inaccuracies at boundaries.
if (absNx > absNy && absNx > absNz) { // Normal is mostly X-axis
(local Y-Z plane)
    u = localPoint.y * uvScale + Ray.EPSILON;
}

```

```

    v = localPoint.z * uvScale + Ray.EPSILON;
} else if (absNy > absNx && absNy > absNz) { // Normal is mostly Y-
axis (local X-Z plane)
    u = localPoint.x * uvScale + Ray.EPSILON;
    v = localPoint.z * uvScale + Ray.EPSILON;
} else { // Normal is mostly Z-axis (local X-Y plane) or equally
dominant
    u = localPoint.x * uvScale + Ray.EPSILON;
    v = localPoint.y * uvScale + Ray.EPSILON;
}

```

// 3. Sample the normal map at the calculated UV coordinates using
ImageTexture's method

```

Color normalMapColor = normalMap.getColorFromUV(u, v); // ***
Düzeltilen kısım burası ***

```

// Convert RGB color from normal map to a tangent-space vector (range
-1 to 1)

// Normal maps typically store (X, Y, Z) components as (R, G, B) where
R,G,B are 0-255.

```

// Convert to -1 to 1 range: (value / 255.0) * 2.0 - 1.0
Vector3 tangentSpaceNormal = new Vector3(
    (normalMapColor.getRed() / 255.0) * 2.0 - 1.0,
    (normalMapColor.getGreen() / 255.0) * 2.0 - 1.0,
    (normalMapColor.getBlue() / 255.0) * 2.0 - 1.0
).normalize();

```

// 4. Calculate Tangent and Bitangent vectors in world space

// These form the basis for the TBN matrix.

// A robust way to get a tangent: cross with a non-parallel axis.

Vector3 tangent;

// Ensure worldNormal is normalized before cross product

Vector3 normalizedWorldNormal = worldNormal.normalize();

```

if (Math.abs(normalizedWorldNormal.x) < 0.9 &&
Math.abs(normalizedWorldNormal.y) < 0.9) {

```

```

    tangent = normalizedWorldNormal.cross(new Vector3(0, 0,
1)).normalize();
} else {

```

```

        tangent = normalizedWorldNormal.cross(new Vector3(0, 1,
0)).normalize();
    }

    Vector3 bitangent =
normalizedWorldNormal.cross(tangent).normalize();

    // 5. Construct the TBN (Tangent, Bitangent, Normal) matrix
    // This matrix transforms a vector from tangent space to world space.
    Matrix3 TBN = new Matrix3(
        tangent.x, bitangent.x, normalizedWorldNormal.x,
        tangent.y, bitangent.y, normalizedWorldNormal.y,
        tangent.z, bitangent.z, normalizedWorldNormal.z
    );

    // 6. Transform the sampled normal from tangent space to world space
    Vector3 worldSpacePerturbedNormal =
TBN.transform(tangentSpaceNormal).normalize();

    // 7. Interpolate between the original geometric normal and the
    perturbed normal based on strength
    // Using Vector3.lerp method (assuming it's added to Vector3 class)
    Vector3 finalNormal = worldNormal.lerp(worldSpacePerturbedNormal,
strength).normalize();

    // 8. Delegate to the base material for actual color calculation using the
    perturbed normal
    return baseMaterial.getColorAt(worldPoint, finalNormal, light,
viewerPos);
}

/**
 * Returns the reflectivity coefficient, delegated to the base material.
 * @return The reflectivity value from the base material.
 */
@Override
public double getReflectivity() {
    return baseMaterial.getReflectivity();
}

```

```
/**  
 * Returns the index of refraction (IOR), delegated to the base material.  
 * @return The IOR value from the base material.  
 */  
@Override  
public double getIndexOfRefraction() {  
    return baseMaterial.getIndexOfRefraction();  
}  
  
/**  
 * Returns the transparency coefficient, delegated to the base material.  
 * @return The transparency value from the base material.  
 */  
@Override  
public double getTransparency() {  
    return baseMaterial.getTransparency();  
}  
}
```

```
// ======  
// File: /net/elenamurat/material/ElenaTextureMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
import java.awt.image.BufferedImage;  
import java.io.File;  
import java.io.IOException;  
import javax.imageio.ImageIO;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.*;  
import net.elena.murat.util.ColorUtil;  
  
public class ElenaTextureMaterial implements Material {
```

```
private final BufferedImage texture;
private Matrix4 objectInverseTransform;

// Material properties
private final double ambientCoeff;
private final double diffuseCoeff;
private final double specularCoeff;
private final double shininess;
private final Color specularColor;
private final double reflectivity;
private final double ior;
private final double transparency;

public ElenaTextureMaterial(String imagePath, Matrix4
objectInverseTransform) throws IOException {
    this(imagePath, 0.1, 0.7, 0.2, 10.0, Color.WHITE, 0.05, 1.0, 0.0,
objectInverseTransform);
}
```

```
public ElenaTextureMaterial(String imagePath,
double ambientCoeff, double diffuseCoeff, double specularCoeff,
double shininess, Color specularColor,
double reflectivity, double ior, double transparency,
Matrix4 objectInverseTransform) throws IOException {
this.texture = ImageIO.read(new File(imagePath));
this.objectInverseTransform = objectInverseTransform;
this.ambientCoeff = ambientCoeff;
this.diffuseCoeff = diffuseCoeff;
this.specularCoeff = specularCoeff;
this.shininess = shininess;
this.specularColor = specularColor;
this.reflectivity = reflectivity;
this.ior = ior;
this.transparency = transparency;
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4();
```

```

        this.objectInverseTransform = tm;
    }

    @Override
    public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
        // 1. Get texture color
        Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
        Color textureColor = getSphericalTextureColor(localPoint);

        // 2. Handle light properties
        LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
        if (props == null) return textureColor;

        // 3. Calculate Phong components
        Color ambient = ColorUtil.multiplyColors(textureColor, props.color,
ambientCoeff);

        if (light instanceof ElenaMuratAmbientLight) {
            return ambient;
        }

        double NdotL = Math.max(0, worldNormal.dot(props.direction));
        Color diffuse = ColorUtil.multiplyColors(textureColor, props.color,
diffuseCoeff * NdotL * props.intensity);

        Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
        Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
        double RdotV = Math.max(0, reflectDir.dot(viewDir));
        double specFactor = Math.pow(RdotV, shininess) * props.intensity;
        Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);

        return ColorUtil.combineColors(ambient, diffuse, specular);
    }

    private Color getSphericalTextureColor(Point3 localPoint) {

```

```

Vector3 normal = localPoint.toVector3().normalize();

// Spherical mapping
double u = 0.5 + Math.atan2(normal.z, normal.x) / (2 * Math.PI);
double v = 0.5 - Math.asin(normal.y) / Math.PI;

// Bilinear sampling
return sampleTexture(u, v);
}

private Color sampleTexture(double u, double v) {
    double x = u * (texture.getWidth() - 1);
    double y = v * (texture.getHeight() - 1);

    int x0 = (int)Math.floor(x);
    int y0 = (int)Math.floor(y);
    int x1 = Math.min(x0 + 1, texture.getWidth() - 1);
    int y1 = Math.min(y0 + 1, texture.getHeight() - 1);

    double fracX = x - x0;
    double fracY = y - y0;

    Color c00 = new Color(texture.getRGB(x0, y0));
    Color c10 = new Color(texture.getRGB(x1, y0));
    Color c01 = new Color(texture.getRGB(x0, y1));
    Color c11 = new Color(texture.getRGB(x1, y1));

    return ColorUtil.bilinearInterpolate(c00, c10, c01, c11, fracX, fracY);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }
}

/**
try {
    Material elenaMat = new ElenaTextureMaterial(
        "textures/elena.png",

```

```
sphere.getInverseTransform()
);
sphere.setMaterial(elenaMat);
} catch (IOException e) {
System.err.println("Texture loading error: " + e.getMessage());
sphere.setMaterial(new DiffuseMaterial(Color.PINK)); // Fallback
}
*/
```

```
// =====
// File: /net/elenamurat/material/AuroraCeramicMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class AuroraCeramicMaterial implements Material {
    private final Color baseColor;
    private final Color auroraColor;
    private final double auroraIntensity;
    private Matrix4 objectTransform;

    private final double ambientCoeff = 0.5;
    private final double diffuseCoeff = 0.7;
    private final double specularCoeff = 0.35;
    private final double shininess = 55.0;
    private final double reflectivity = 0.22;
    private final double ior = 1.7;
    private final double transparency = 0.0;

    public AuroraCeramicMaterial() {
        this(new Color(0xF5, 0xF5, 0xDC), new Color(0x00, 0xFF, 0x7F),
0.45);
```

```
}
```

```
public AuroraCeramicMaterial(Color baseColor, Color auroraColor,  
double auroraIntensity) {  
    this.baseColor = baseColor;  
    this.auroraColor = auroraColor;  
    this.auroraIntensity = Math.max(0, Math.min(1, auroraIntensity));  
    this.objectTransform = Matrix4.identity();  
}
```

```
@Override
```

```
public void setObjectTransform(Matrix4 tm) {  
    if (tm == null) tm = new Matrix4();  
    this.objectTransform = tm;  
}
```

```
@Override
```

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light  
light, Point3 viewerPos) {
```

```
    Point3 objectPoint =  
    objectTransform.inverse().transformPoint(worldPoint);
```

```
    Color surfaceColor = calculateAuroraEffect(objectPoint, worldNormal,  
viewerPos);
```

```
    LightProperties props = LightProperties.getLightProperties(light,  
worldPoint);
```

```
    if (props == null) return surfaceColor;
```

```
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,  
ambientCoeff);
```

```
    if (light instanceof ElenaMuratAmbientLight) {  
        return ambient;  
    }
```

```
    double NdotL = Math.max(0, worldNormal.dot(props.direction));  
    Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,  
diffuseCoeff * NdotL * props.intensity);
```

```

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(Color.WHITE, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

private Color calculateAuroraEffect(Point3 point, Vector3 normal, Point3
viewerPos) {
    double x = point.x * 7.0;
    double y = point.y * 7.0;
    double z = point.z * 7.0;

    // Aurora-like flowing patterns
    double flow1 = Math.sin(x * 1.2 + Math.cos(y * 0.8) + Math.sin(z *
1.5));
    double flow2 = Math.cos(x * 0.7 + Math.sin(y * 1.3) + Math.cos(z *
0.9));
    double flow3 = Math.sin(x * 2.1 + y * 1.7 + z * 0.5);

    double auroraPattern = (flow1 * 0.4 + flow2 * 0.3 + flow3 * 0.3);
    double normalizedPattern = (auroraPattern + 1.0) * 0.5;

    // View-dependent effect for aurora
    Vector3 viewDir = viewerPos.subtract(point).normalize();
    double viewAngle = Math.abs(viewDir.dot(normal));
    double viewEffect = Math.pow(viewAngle, 0.7);

    if (normalizedPattern < auroraIntensity) {
        // Aurora glow effect
        double intensity = normalizedPattern / auroraIntensity * viewEffect;
        return createAuroraGlow(baseColor, auroraColor, intensity);
    } else {
        // Ceramic base with subtle glow
        double intensity = (normalizedPattern - auroraIntensity) / (1.0 -

```

```

auroraIntensity);
    return addCeramicGlow(baseColor, auroraColor, intensity * 0.3 *
viewEffect);
}
}

private Color createAuroraGlow(Color base, Color glow, double
intensity) {
    int r = (int)(base.getRed() * (1-intensity) + glow.getRed() * intensity);
    int g = (int)(base.getGreen() * (1-intensity) + glow.getGreen() *
intensity);
    int b = (int)(base.getBlue() * (1-intensity) + glow.getBlue() * intensity);

    // Add extra glow effect
    double glowBoost = intensity * 0.5;
    r = Math.min(255, r + (int)(glow.getRed() * glowBoost));
    g = Math.min(255, g + (int)(glow.getGreen() * glowBoost));
    b = Math.min(255, b + (int)(glow.getBlue() * glowBoost));

    return new Color(r, g, b);
}

private Color addCeramicGlow(Color base, Color glow, double intensity)
{
    int r = (int)(base.getRed() + glow.getRed() * intensity * 0.3);
    int g = (int)(base.getGreen() + glow.getGreen() * intensity * 0.4);
    int b = (int)(base.getBlue() + glow.getBlue() * intensity * 0.2);

    return new Color(
        Math.min(255, Math.max(0, r)),
        Math.min(255, Math.max(0, g)),
        Math.min(255, Math.max(0, b)))
};

}

@Override
public double getReflectivity() {
    return reflectivity;
}

```

```
@Override  
public double getIndexOfRefraction() {  
    return ior;  
}  
  
@Override  
public double getTransparency() {  
    return transparency;  
}  
}  
  
// ======  
// File: /net/lena/murat/material/ProceduralFlowerMaterial.java  
// ======
```

```
package net.elena.murat.material;  
  
import java.awt.Color;  
  
import net.elena.murat.math.*;  
import net.elena.murat.light.*;  
  
public class ProceduralFlowerMaterial implements Material {  
    private final double petalCount;  
    private final Color petalColor;  
    private final Color centerColor;  
    private final double ambientStrength;  
    private final double reflectivity = 0.1;  
  
    public ProceduralFlowerMaterial(double petalCount, Color petalColor,  
        Color centerColor) {  
        this(petalCount, petalColor, centerColor, 0.2);  
    }  
  
    public ProceduralFlowerMaterial(double petalCount, Color petalColor,  
        Color centerColor, double ambientStrength) {
```

```

this.petalCount = petalCount;
this.petalColor = petalColor;
this.centerColor = centerColor;
this.ambientStrength = Math.max(0, Math.min(1, ambientStrength));
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 normal, Light light,
Point3 viewPos) {
    // Light information
    Color lightColor = light.getColor();
    double intensity = light.getIntensityAt(worldPoint);
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

    // Diffuse shading
    double NdotL = Math.max(0, normal.dot(lightDir));
    double diffuseFactor = NdotL * intensity;

    // Polar coordinates
    double u = worldPoint.x / 2.0;
    double v = worldPoint.z / 2.0;
    double r = Math.sqrt(u*u + v*v);
    double theta = Math.atan2(v, u);
    double flowerR = 0.3 * Math.cos(petalCount * theta);

    // Color selection + light effect
    Color baseColor;
    if (r < 0.15) {
        baseColor = centerColor; // Center
    }
    else if (Math.abs(r - flowerR) < 0.05) {
        baseColor = petalColor; // Petals
    }
    else if (r < 0.6 && Math.random() < 0.3) {
        baseColor = new Color(0, 100 + (int)(155 * Math.random()), 0); // Green leaves
    }
    else {
        baseColor = Color.WHITE; // Background
    }
}

```

```

}

// Light color blending
return applyLight(baseColor, lightColor, diffuseFactor);
}

private Color applyLight(Color baseColor, Color lightColor, double
diffuseFactor) {
    // Ambient component
    int ambientR = (int)(baseColor.getRed() * ambientStrength);
    int ambientG = (int)(baseColor.getGreen() * ambientStrength);
    int ambientB = (int)(baseColor.getBlue() * ambientStrength);

    // Diffuse + Specular (multiplied by light color)
    int r = (int)(baseColor.getRed() * (lightColor.getRed()/255.0) *
diffuseFactor);
    int g = (int)(baseColor.getGreen() * (lightColor.getGreen()/255.0) *
diffuseFactor);
    int b = (int)(baseColor.getBlue() * (lightColor.getBlue()/255.0) *
diffuseFactor);

    // Final color (ambient + diffuse)
    r = Math.min(255, ambientR + r);
    g = Math.min(255, ambientG + g);
    b = Math.min(255, ambientB + b);

    return new Color(r, g, b);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return 1.0; }
@Override public double getTransparency() { return 0.0; }

@Override
public void setObjectTransform(Matrix4 tm) {
}

}

```

```

/***
// Adding to scene
Plane flowerPlane = new Plane(new Point3(0,0,0), new Vector3(0,1,0));
Material flowerMat = new ProceduralFlowerMaterial(
    5,          // 5 petals
    Color.RED,   // Petal color
    Color.YELLOW // Center color
);
flowerPlane.setMaterial(flowerMat);

// Camera setup
rayTracer.setCameraPosition(new Point3(0, 2, 3));
*/

```

```

// =====
// File: /net/elenamurat/material/AnodizedMetalMaterial.java
// =====

```

```

// AnodizedMetalMaterial.java - CHECKERBOARD STYLE
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;
import net.elena.murat.util.ColorUtil;

public class AnodizedMetalMaterial implements Material {
    private final Color baseColor;
    private Matrix4 objectTransform;

    // Phong parameters optimized for metallic surface
    private final double ambientCoeff = 0.3;
    private final double diffuseCoeff = 0.2; // Metallic surfaces have weak
    diffuse
    private final double specularCoeff = 1.0; // Strong specular for metallic
    private final double shininess = 100.0;
    private final Color specularColor = Color.WHITE;
}
```

```
private final double reflectivity = 0.8;
private final double ior = 2.4;
private final double transparency = 0.0;

public AnodizedMetalMaterial() {
    this(new Color(50, 50, 200));
}

public AnodizedMetalMaterial(Color baseColor) {
    this.baseColor = baseColor;
    this.objectTransform = Matrix4.identity();
}

@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectTransform = tm;
}

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    // 1. Get base color with iridescence effect
    Color surfaceColor = calculateIridescentColor(worldPoint,
worldNormal, viewerPos);

    // 2. Handle light properties (EXACTLY like Checkerboard)
    LightProperties props = LightProperties.getLightProperties(light,
worldPoint);
    if (props == null) return surfaceColor;

    // 3. Calculate Phong components (EXACTLY like Checkerboard)
    Color ambient = ColorUtil.multiplyColors(surfaceColor, props.color,
ambientCoeff);

    if (light instanceof ElenaMuratAmbientLight) {
        return ambient;
    }
}
```

```

double NdotL = Math.max(0, worldNormal.dot(props.direction));
Color diffuse = ColorUtil.multiplyColors(surfaceColor, props.color,
diffuseCoeff * NdotL * props.intensity);

Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = props.direction.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess) * props.intensity;
Color specular = ColorUtil.multiplyColors(specularColor, props.color,
specularCoeff * specFactor);

return ColorUtil.combineColors(ambient, diffuse, specular);
}

```

```

private Color calculateIridescentColor(Point3 worldPoint, Vector3
normal, Point3 viewerPos) {
    // Calculate view angle for iridescence effect
    Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
    double viewAngle = Math.abs(viewDir.dot(normal));

    // Simple iridescence effect based on view angle
    double hueShift = viewAngle * 180.0;

    int r = baseColor.getRed();
    int g = baseColor.getGreen();
    int b = baseColor.getBlue();

    // Blue -> Purple -> Pink transition (anodized aluminum effect)
    if (viewAngle < 0.3) {
        // Narrow angle - blue tones
        r = (int)(r * 0.7);
        g = (int)(g * 0.8);
        b = (int)(b * 1.2);
    } else if (viewAngle < 0.6) {
        // Medium angle - purple tones
        r = (int)(r * 1.1);
        g = (int)(g * 0.7);
        b = (int)(b * 1.0);
    } else {

```

```

// Wide angle - pink/gold tones
r = (int)(r * 1.3);
g = (int)(g * 0.9);
b = (int)(b * 0.8);
}

return new Color(
    Math.min(255, Math.max(0, r)),
    Math.min(255, Math.max(0, g)),
    Math.min(255, Math.max(0, b)))
);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }

}

```

```

// =====
// File: /net/elenamurat/material/InvertLightColorMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.awt.Color;
import java.util.ArrayList;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.light.Light;
import net.elena.murat.util.ColorUtil;

```

```

public class InvertLightColorMaterial implements Material {

```

```

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPoint) {

```

```
Color originalColor = light.getColor();

int invertedRed = 255 - originalColor.getRed();
int invertedGreen = 255 - originalColor.getGreen();
int invertedBlue = 255 - originalColor.getBlue();

return new Color(invertedRed, invertedGreen, invertedBlue);
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
}
```

```
@Override
public double getTransparency() {
    return 0.1;
}
```

```
@Override
public double getReflectivity() {
    return 0.15;
}
```

```
@Override
public double getIndexOfRefraction() {
    return 1.0;
}
```

```
}
```

```
// =====
// File: /net/elenamurat/material/EmissiveMaterial.java
// =====
```

```
package net.elena.murat.material;
```

```
import java.awt.Color;
```

```
import net.elena.murat.math.*;
import net.elena.murat.light.Light;

/**
 * EmissiveMaterial represents a surface that emits its own light, acting as
 * a light source itself.
 * It does not reflect or refract light from other sources.
 * This material ignores all external lights since it's self-illuminating.
 */

public class EmissiveMaterial implements Material {
    private final Color emissiveColor;
    private final double emissiveStrength;

    // Material properties constants
    private static final double REFLECTIVITY = 0.0;
    private static final double IOR = 1.0;
    private static final double TRANSPARENCY = 0.0;

    public EmissiveMaterial(Color emissiveColor, double emissiveStrength)
    {
        this.emissiveColor = new Color(
            emissiveColor.getRed(),
            emissiveColor.getGreen(),
            emissiveColor.getBlue()
        );
        this.emissiveStrength = Math.max(0, emissiveStrength);
    }

    public Color getEmissiveColor() {
        return new Color(
            emissiveColor.getRed(),
            emissiveColor.getGreen(),
            emissiveColor.getBlue()
        );
    }

    public double getEmissiveStrength() {
        return emissiveStrength;
    }
}
```

```
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // Emissive materials ignore all external lights and viewer position
    int r = clampColorValue(emissiveColor.getRed() * emissiveStrength);
    int g = clampColorValue(emissiveColor.getGreen() *
emissiveStrength);
    int b = clampColorValue(emissiveColor.getBlue() * emissiveStrength);

    return new Color(r, g, b);
}

@Override
public double getReflectivity() {
    return REFLECTIVITY;
}

@Override
public double getIndexOfRefraction() {
    return IOR;
}

@Override
public double getTransparency() {
    return TRANSPARENCY;
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

private int clampColorValue(double value) {
    return (int) Math.min(255, Math.max(0, value));
}
```

```
// =====
// File: /net/elenamurat/material/OpticalIllusionMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

/**
 * Optical Illusion Material
 */
public class OpticalIllusionMaterial implements Material {
    private final Color color1;
    private final Color color2;
    private final double frequency;
    private final double smoothness;
    private Matrix4 objectInverseTransform;

    private final double ambientCoefficient;
    private final double diffuseCoefficient;
    private final double specularCoefficient;
    private final double shininess;
    private final double reflectivity;
    private final double ior;
    private final double transparency;

    public OpticalIllusionMaterial(Color color1, Color color2,
        double frequency, double smoothness,
        double ambient, double diffuse, double specular,
        double shininess, double reflectivity,
        double ior, double transparency,
        Matrix4 objectInverseTransform) {
        this.color1 = color1;
        this.color2 = color2;
        this.frequency = Math.max(0.1, frequency);
        this.smoothness = Math.max(0, Math.min(1, smoothness));
    }
}
```

```
this.ambientCoefficient = Math.max(0, Math.min(1, ambient));
this.diffuseCoefficient = Math.max(0, Math.min(1, diffuse));
this.specularCoefficient = Math.max(0, Math.min(1, specular));
this.shininess = Math.max(1, shininess);
this.reflectivity = Math.max(0, Math.min(1, reflectivity));
this.ior = ior;
this.transparency = Math.max(0, Math.min(1, transparency));
this.objectInverseTransform = objectInverseTransform;
}
```

```
public OpticalIllusionMaterial(Color color1, Color color2,
    double frequency, double smoothness,
    Matrix4 objectInverseTransform) {
    this(color1, color2, frequency, smoothness,
        0.15, 0.6, 0.25, 30.0, 0.05, 1.2, 0.02,
        objectInverseTransform);
}
```

```
@Override
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}

private Color getPatternColor(double u, double v) {
    double dist = Math.sqrt(u*u + v*v);
    double pattern = 0.5 * (1 + Math.sin(dist * frequency * 2 * Math.PI));

    if (smoothness > 0) {
        pattern = smoothstep(0.5-smoothness, 0.5+smoothness, pattern);
    }

    return blendColors(color1, color2, pattern);
}
```

```
private double smoothstep(double edge0, double edge1, double x) {
    x = Math.max(0, Math.min(1, (x - edge0) / (edge1 - edge0)));
    return x * x * (3 - 2 * x);
}
```

```

@Override
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        return Color.BLACK;
    }

    // UV mapping
    Point3 localPoint =
    objectInverseTransform.transformPoint(worldPoint);
    Vector3 localNormal =
    objectInverseTransform.inverseTransposeForNormal()
    .transformVector(worldNormal).normalize();

    double u, v;
    double absX = Math.abs(localNormal.x);
    double absY = Math.abs(localNormal.y);
    double absZ = Math.abs(localNormal.z);

    if (absX > absY && absX > absZ) {
        u = localPoint.y;
        v = localPoint.z;
    } else if (absY > absX && absY > absZ) {
        u = localPoint.x;
        v = localPoint.z;
    } else {
        u = localPoint.x;
        v = localPoint.y;
    }

    Color baseColor = getPatternColor(u, v);
    Color lightColor = light.getColor();
    double intensity = light.getIntensity();
    Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

    // 1. Ambient (light color included)
    int r = (int)(baseColor.getRed() * ambientCoefficient *
(lightColor.getRed()/255.0));

```

```

int g = (int)(baseColor.getGreen() * ambientCoefficient *
(lightColor.getGreen()/255.0));
int b = (int)(baseColor.getBlue() * ambientCoefficient *
(lightColor.getBlue()/255.0));

// 2. Diffuse (light color included)
double NdotL = Math.max(0, worldNormal.dot(lightDir));
if (NdotL > 0) {
    r += (int)(baseColor.getRed() * diffuseCoefficient * NdotL *
(lightColor.getRed()/255.0) * intensity);
    g += (int)(baseColor.getGreen() * diffuseCoefficient * NdotL *
(lightColor.getGreen()/255.0) * intensity);
    b += (int)(baseColor.getBlue() * diffuseCoefficient * NdotL *
(lightColor.getBlue()/255.0) * intensity);

// 3. Specular (light color included)
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specular = Math.pow(RdotV, shininess) * specularCoefficient *
intensity;

r += (int)(255 * specular * (lightColor.getRed()/255.0));
g += (int)(255 * specular * (lightColor.getGreen()/255.0));
b += (int)(255 * specular * (lightColor.getBlue()/255.0));
}

return new Color(
    Math.min(255, r),
    Math.min(255, g),
    Math.min(255, b)
);
}

```

```

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }

```

```
private Color blendColors(Color c1, Color c2, double ratio) {
```

```

ratio = Math.max(0, Math.min(1, ratio));
int r = (int)(c1.getRed() * (1-ratio) + c2.getRed() * ratio);
int g = (int)(c1.getGreen() * (1-ratio) + c2.getGreen() * ratio);
int b = (int)(c1.getBlue() * (1-ratio) + c2.getBlue() * ratio);
return new Color(Math.min (255, r),
                 Math.min (255, g),
                 Math.min (255, b));
}

} // ***

OpticalIllusionMaterial illusionMat = new OpticalIllusionMaterial(
Color.BLACK,
Color.WHITE,
5.0, // Ring frequency
0.2, // Transition smoothness
object.getInverseTransform()
);

OpticalIllusionMaterial hypnoMat = new OpticalIllusionMaterial(
new Color(0, 100, 255), // Blue
new Color(255, 50, 0), // Orange
8.0, // More frequent rings
0.1, // Sharp transitions
object.getInverseTransform()
);
*/

```

```

// =====
// File: /net/elenamurat/material/MetallicMaterial.java
// =====

```

```

package net.elena.murat.material;

import java.lang.reflect.Method;

import java.awt.Color;

```

```

import net.elena.murat.math.*;
import net.elena.murat.light.*;

/**
 * MetallicMaterial represents a metallic surface with strong, colored
 * specular highlights
 * and typically low diffuse reflection. It is designed to work with ray
 * tracing
 * that handles reflections recursively.
 * This material now fully implements the extended Material interface.
 */

public class MetallicMaterial implements Material {
    private Color metallicColor; // The base color of the metal
    private Color specularColor; // The color of the specular highlight (can be
metallicColor for true metals)
    private double reflectivity; // How much light is reflected (0.0 to 1.0)
    private double shininess; // Shininess exponent for specular highlights
    private double ambientCoefficient;
    private double diffuseCoefficient;
    private double specularCoefficient;

    // Default values for Material interface methods, as this material is
opaque and not refractive
    private final double ior = 1.0; // Index of Refraction for air/vacuum
    private final double transparency = 0.0; // Not transparent

    /**
     * Constructs a MetallicMaterial with specified properties.
     * @param metallicColor The base color of the metal.
     * @param specularColor The color of the specular highlights. For true
metals, this is often the same as metallicColor.
     * @param reflectivity The reflectivity coefficient (0.0 to 1.0).
     * @param shininess The shininess exponent for specular highlights.
     * @param ambientCoefficient The ambient light contribution coefficient
(0.0 to 1.0).
     * @param diffuseCoefficient The diffuse light contribution coefficient
(0.0 to 1.0).
     * @param specularCoefficient The specular light contribution
coefficient (0.0 to 1.0).
    */
}

```

```
/*
public MetallicMaterial(Color metallicColor, Color specularColor,
double reflectivity, double shininess,
    double ambientCoefficient, double diffuseCoefficient, double
specularCoefficient) {
    this.metallicColor = metallicColor;
    this.specularColor = specularColor;
    this.reflectivity = reflectivity;
    this.shininess = shininess;
    this.ambientCoefficient = ambientCoefficient;
    this.diffuseCoefficient = diffuseCoefficient;
    this.specularCoefficient = specularCoefficient;
}
```

```
// Getter methods for material properties
public Color getMetallicColor() { return metallicColor; }
public Color getSpecularColor() { return specularColor; }
public double getShininess() { return shininess; } // Not part of Material
interface, but useful internally
public double getAmbientCoefficient() { return ambientCoefficient; } // // Not part of Material interface
public double getDiffuseCoefficient() { return diffuseCoefficient; } // Not
part of Material interface
public double getSpecularCoefficient() { return specularCoefficient; } // // Not part of Material interface
```

```
/**
```

- * Calculates the direct lighting color at a given point on the metallic surface.
- * This method computes the ambient, diffuse (typically low for metals), and specular components.
- * The actual reflection will be handled by the ray tracer's recursive shading process,
- * using the `getReflectivity()` value.
- *
- * @param point The intersection point in world coordinates.
- * @param normal The surface normal at the intersection point in world coordinates.
- * @param light The single light source to be used in the calculation.

```

* @param viewerPos The position of the viewer (camera) in world
coordinates.
* @return The calculated color contribution from this light source.
*/
@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    Color lightColor = light.getColor();
    double attenuatedIntensity = 0.0; // Initialize for non-ambient lights

    // Ambient component
    int rAmbient = (int) (metallicColor.getRed() * ambientCoefficient *
lightColor.getRed() / 255.0);
    int gAmbient = (int) (metallicColor.getGreen() * ambientCoefficient *
lightColor.getGreen() / 255.0);
    int bAmbient = (int) (metallicColor.getBlue() * ambientCoefficient *
lightColor.getBlue() / 255.0);

    if (light instanceof ElenaMuratAmbientLight) {
        return new Color(
            Math.min(255, rAmbient),
            Math.min(255, gAmbient),
            Math.min(255, bAmbient)
        );
    }

    Vector3 lightDirection = getLightDirection(light, point);
    if (lightDirection == null) { // Handle unsupported light types
        return Color.BLACK;
    }

    // Get attenuated intensity based on light type
    if (light instanceof MuratPointLight) {
        attenuatedIntensity = ((MuratPointLight)
light).getAttenuatedIntensity(point);
    } else if (light instanceof ElenaDirectionalLight) {
        attenuatedIntensity = ((ElenaDirectionalLight) light).getIntensity();
    } else if (light instanceof PulsatingPointLight) {
        attenuatedIntensity = ((PulsatingPointLight)

```

```

light).getAttenuatedIntensity(point);
} else if (light instanceof SpotLight) {
attenuatedIntensity = ((SpotLight) light).getAttenuatedIntensity(point);
} else if (light instanceof BioluminescentLight) {
attenuatedIntensity = ((BioluminescentLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof BlackHoleLight) {
attenuatedIntensity = ((BlackHoleLight)
light).getAttenuatedIntensity(point);
} else if (light instanceof FractalLight) {
attenuatedIntensity = ((FractalLight)
light).getAttenuatedIntensity(point);
} else {
System.err.println("Warning: Unknown or unsupported light type for
MetallicMaterial shading (intensity): " + light.getClass().getName());
return Color.BLACK;
}

// Diffuse component (typically very low for metals)
double NdotL = Math.max(0, normal.dot(lightDirection));
int rDiffuse = (int) (metallicColor.getRed() * diffuseCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * NdotL);
int gDiffuse = (int) (metallicColor.getGreen() * diffuseCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * NdotL);
int bDiffuse = (int) (metallicColor.getBlue() * diffuseCoefficient *
lightColor.getBlue() / 255.0 * attenuatedIntensity * NdotL);

// Specular component (strong and colored for metals)
Vector3 viewDir = viewerPos.subtract(point).normalize();
Vector3 reflectDir = lightDirection.negate().reflect(normal); // Use
negate() before reflect() for correct reflection vector
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specFactor = Math.pow(RdotV, shininess);

int rSpecular = (int) (specularColor.getRed() * specularCoefficient *
lightColor.getRed() / 255.0 * attenuatedIntensity * specFactor);
int gSpecular = (int) (specularColor.getGreen() * specularCoefficient *
lightColor.getGreen() / 255.0 * attenuatedIntensity * specFactor);
int bSpecular = (int) (specularColor.getBlue() * specularCoefficient *

```

```

lightColor.getBlue() / 255.0 * attenuatedIntensity * specFactor);

    // Sum up all components for this light source
    // Note: Ambient component is added only once per pixel in RayTracer's
    shade method,
    // so here we only return the diffuse and specular contribution for non-
    ambient lights.
    int finalR = Math.min(255, rDiffuse + rSpecular);
    int finalG = Math.min(255, gDiffuse + gSpecular);
    int finalB = Math.min(255, bDiffuse + bSpecular);

    return new Color(finalR, finalG, finalB);
}

/***
 * Returns the reflectivity coefficient of the material.
 * @return The reflectivity value (0.0-1.0).
 */
@Override
public double getReflectivity() { return reflectivity; }

/***
 * Returns the index of refraction (IOR) of the material. For opaque
metallic materials, this is typically 1.0.
 * @return 1.0
 */
@Override
public double getIndexOfRefraction() { return ior; }

/***
 * Returns the transparency coefficient of the material. For opaque
metallic materials, this is typically 0.0.
 * @return 0.0
 */
@Override
public double getTransparency() { return transparency; }

@Override
public void setObjectTransform(Matrix4 tm) {

```

```

}

/***
 * Helper method to get the normalized light direction vector from a light
source to a point.
 * @param light The light source.
 * @param point The point in world coordinates.
 * @return Normalized light direction vector, or null if light type is
unsupported.
 */

private Vector3 getLightDirection(Light light, Point3 point) {
    final Vector3 safeVector=new Vector3 (0, 1, 0).normalize ();

    if (light == null) {
        return safeVector;// Varsayılan yön
    }

    if (light instanceof MuratPointLight) {
        return
    ((MuratPointLight)light).getPosition().subtract(point).normalize();
    } else if (light instanceof ElenaDirectionalLight) {
        return
    ((ElenaDirectionalLight)light).getDirection().negate().normalize();
    } else if (light instanceof PulsatingPointLight) {
        return
    ((PulsatingPointLight)light).getPosition().subtract(point).normalize();
    } else if (light instanceof SpotLight) {
        return ((SpotLight)light).getDirectionAt(point).normalize();
    } else if (light instanceof BioluminescentLight) {
        return ((BioluminescentLight)light).getDirectionAt(point);
    } else if (light instanceof BlackHoleLight) {
        return ((BlackHoleLight)light).getDirectionAt(point);
    } else if (light instanceof FractalLight) {
        return ((FractalLight)light).getDirectionAt(point);
    } else {
        //return safeVector;
    }
}

```

```
try {
    Method getDirMethod = light.getClass().getMethod("getDirectionAt",
Point3.class);
    return (Vector3) getDirMethod.invoke(light, point);
}
catch (Exception e) {
    System.err.println("Unsupported light type: " +
light.getClass().getName());
    return new Vector3(0, 1, 0).normalize(); // Güvenli varsayılan
}
}
```

```
// =====
// File: /net/elenamurat/material/NeutralMaterial.java
// =====
```

```
package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.light.Light;
import net.elena.murat.math.*;

/**
 * Neutral material that shows the original pixel colors without any
lighting effects
 */
public class NeutralMaterial implements Material {
    private final Color baseColor;
    private final double reflectivity;
    private final double transparency;
    private final double indexOfRefraction;

    public NeutralMaterial(Color baseColor) {
        this(baseColor, 0.0, 0.0, 1.0);
    }
```

```
public NeutralMaterial(Color baseColor, double reflectivity,
    double transparency, double indexOfRefraction) {
    this.baseColor = baseColor;
    this.reflectivity = reflectivity;
    this.transparency = transparency;
    this.indexOfRefraction = indexOfRefraction;
}

@Override
public Color getColorAt(Point3 point, Vector3 normal, Light light,
Point3 viewerPos) {
    // Return base color without any lighting calculations
    return baseColor;
}

@Override
public double getReflectivity() {
    return reflectivity;
}

@Override
public double getIndexOfRefraction() {
    return indexOfRefraction;
}

@Override
public double getTransparency() {
    return transparency;
}

@Override
public void setObjectTransform(Matrix4 tm) {
}

// =====
```

```
// File: /net/elenamurat/material/PixelArtMaterial.java
// =====

package net.elena.murat.material;

import java.awt.Color;

import net.elena.murat.math.*;
import net.elena.murat.light.*;

/**
 * Pixel art material with correct lights
 */
public class PixelArtMaterial implements Material {
    private final Color[] palette;
    private final double pixelSize;
    private Matrix4 objectInverseTransform;

    private final double ambientCoefficient;
    private final double diffuseCoefficient;
    private final double specularCoefficient;
    private final double shininess;
    private final double reflectivity;
    private final double transparency;
    private final double ior;

    public PixelArtMaterial(Color[] palette, double pixelSize,
                           double ambient, double diffuse, double specular,
                           double shininess, double reflectivity,
                           double ior, double transparency,
                           Matrix4 objectInverseTransform) {
        this.palette = palette.length >= 2 ? palette :
            new Color[]{Color.RED, Color.BLUE};
        this.pixelSize = Math.max(0.01, pixelSize);
        this.ambientCoefficient = clamp(ambient);
        this.diffuseCoefficient = clamp(diffuse);
        this.specularCoefficient = clamp(specular);
        this.shininess = Math.max(1, shininess);
        this.reflectivity = clamp(reflectivity);
    }
}
```

```
this.ior = ior;
this.transparency = clamp(transparency);
this.objectInverseTransform = objectInverseTransform;
}
```

```
public PixelArtMaterial(Color[] palette, double pixelSize,
Matrix4 objectInverseTransform) {
    this(palette, pixelSize, 0.1, 0.7, 0.2, 10.0, 0.0, 1.0, 0.0,
objectInverseTransform);
}
```

@Override

```
public void setObjectTransform(Matrix4 tm) {
    if (tm == null) tm = new Matrix4 ();
    this.objectInverseTransform = tm;
}
```

```
private double clamp(double value) {
    return Math.max(0, Math.min(1, value));
}
```

```
private Color getPixelColor(double u, double v) {
    int x = (int)(u / pixelSize);
    int y = (int)(v / pixelSize);
    int hash = (x * 7919 + y * 7901) % palette.length;
    return palette[Math.abs(hash) % palette.length];
}
```

@Override

```
public Color getColorAt(Point3 worldPoint, Vector3 worldNormal, Light
light, Point3 viewerPos) {
    if (objectInverseTransform == null) {
        return Color.BLACK;
    }
}
```

```
// Transform to object space for UV mapping
Point3 localPoint =
objectInverseTransform.transformPoint(worldPoint);
Vector3 localNormal =
```

```

objectInverseTransform.inverseTransposeForNormal()
    .transformVector(worldNormal).normalize();

// UV mapping by dominant normal axis
double absX = Math.abs(localNormal.x);
double absY = Math.abs(localNormal.y);
double absZ = Math.abs(localNormal.z);

double u, v;
if (absX > absY && absX > absZ) {
    u = localPoint.y;
    v = localPoint.z;
} else if (absY > absX && absY > absZ) {
    u = localPoint.x;
    v = localPoint.z;
} else {
    u = localPoint.x;
    v = localPoint.y;
}

// Base pixel color
Color baseColor = getPixelColor(u, v);

// Light calculations
Color lightColor = light.getColor();
double intensity = light.getIntensity();
Vector3 lightDir = light.getDirectionAt(worldPoint).normalize();

// 1. Ambient component (light color included)
int r = (int)(baseColor.getRed() * ambientCoefficient *
(lightColor.getRed()/255.0));
int g = (int)(baseColor.getGreen() * ambientCoefficient *
(lightColor.getGreen()/255.0));
int b = (int)(baseColor.getBlue() * ambientCoefficient *
(lightColor.getBlue()/255.0));

// 2. Diffuse component (light color included)
double NdotL = Math.max(0, worldNormal.dot(lightDir));
if (NdotL > 0) {

```

```

    r += (int)(baseColor.getRed() * diffuseCoefficient * NdotL *
(lightColor.getRed()/255.0) * intensity);
    g += (int)(baseColor.getGreen() * diffuseCoefficient * NdotL *
(lightColor.getGreen()/255.0) * intensity);
    b += (int)(baseColor.getBlue() * diffuseCoefficient * NdotL *
(lightColor.getBlue()/255.0) * intensity);

// 3. Specular component (light color included)
Vector3 viewDir = viewerPos.subtract(worldPoint).normalize();
Vector3 reflectDir = lightDir.negate().reflect(worldNormal);
double RdotV = Math.max(0, reflectDir.dot(viewDir));
double specular = Math.pow(RdotV, shininess) * specularCoefficient *
intensity;

r += (int)(255 * specular * (lightColor.getRed()/255.0));
g += (int)(255 * specular * (lightColor.getGreen()/255.0));
b += (int)(255 * specular * (lightColor.getBlue()/255.0));
}

return new Color(
    Math.min(255, r),
    Math.min(255, g),
    Math.min(255, b)
);
}

@Override public double getReflectivity() { return reflectivity; }
@Override public double getIndexOfRefraction() { return ior; }
@Override public double getTransparency() { return transparency; }
}

/**
Color[] retroPalette = {
new Color(255, 0, 0), // Red
new Color(0, 255, 0), // Green
new Color(0, 0, 255), // Blue
new Color(255, 255, 0) // Yellow
};
```

```
PixelArtMaterial pixelMat = new PixelArtMaterial(  
retroPalette,  
0.3, // Pixel size  
object.getInverseTransform()  
);
```

```
Color[] gameboyPalette = {  
new Color(15, 56, 15), // Dark green  
new Color(48, 98, 48), // Medium green  
new Color(139, 172, 15), // Light green  
new Color(155, 188, 15) // Highlight green  
};
```

```
PixelArtMaterial gbMat = new PixelArtMaterial(  
gameboyPalette,  
0.5, // Larger pixels  
object.getInverseTransform()  
);
```

```
// Directional Light (Sun)  
Light sun = new DirectionalLight(  
new Vector3(0, -1, 0), // From top to bottom  
Color.WHITE,  
1.5  
);
```

```
// Point Light (Bulb)  
Light bulb = new PointLight(  
new Point3(2, 3, 0), // Position  
Color.YELLOW,  
2.0  
);  
*/
```

```
// ======  
// File: /net/elenamurat/shape/TorusKnot.java  
// ======
```

```

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/**
 * (p,q) type torus knot
 * EMShape arayüzüünü tam olarak uygular
 */

public class TorusKnot implements EMShape {
    private final double R; // Torus major radius
    private final double r; // Knot tube radius
    private final int p; // Knot p parametresi
    private final int q; // Knot q parametresi
    private Material material;
    private Matrix4 transform = Matrix4.identity();
    private Matrix4 inverseTransform = Matrix4.identity();
    private static final double EPSILON = 1e-4;

    public TorusKnot(double R, double r, int p, int q) {
        this.R = Math.max(0.1, R);
        this.r = Math.max(0.05, r);
        this.p = p;
        this.q = q;
    }

    @Override
    public double intersect(Ray ray) {
        Ray localRay = new Ray(
            inverseTransform.transformPoint(ray.getOrigin()),
            inverseTransform.transformVector(ray.getDirection()).normalize()
        );

        // Sphere marching ile kesişim
        double t = 0.0;
        double stepSize = 0.05;
        int maxSteps = 200;
    }
}

```

```

double threshold = 0.005;

for (int i = 0; i < maxSteps; i++) {
    Point3 point = localRay.pointAtParameter(t);
    double dist = signedDistanceFunction(point);

    if (dist < threshold) {
        if (t > Ray.EPSILON) {
            return t;
        }
        break;
    }

    if (t > 100.0) break;
    t += Math.max(dist * 0.5, stepSize);
}

return -1.0;
}

/**
 * Calculates all intersection intervals between a ray and this torus knot
 * using ray marching.
 * Detects both entry (tIn) and exit (tOut) points by monitoring the SDF
 * sign change.
 * @param ray The ray to test, in world coordinates.
 * @return A list of IntersectionInterval objects. Empty if no valid
 * interval.
 */
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize()
    );

    double t = 0.0;
    double stepSize = 0.05;
}

```

```

int maxSteps = 200;
double threshold = 0.005;
double maxDistance = 100.0;

List<IntersectionInterval> intervals = new java.util.ArrayList<>();
boolean isInside = false;
double tIn = -1;

for (int i = 0; i < maxSteps; i++) {
    Point3 point = localRay.pointAtParameter(t);
    double dist = signedDistanceFunction(point);

    boolean wasInside = isInside;
    isInside = dist < threshold;

    // Entry: from outside to inside
    if (!wasInside && isInside && tIn < 0 && t > Ray.EPSILON) {
        tIn = t;
    }

    // Exit: from inside to outside
    if (wasInside && !isInside && tIn >= 0) {
        double tOut = t;

        // Create Intersection objects
        Point3 worldIn = ray.pointAtParameter(tIn);
        Point3 worldOut = ray.pointAtParameter(tOut);
        Vector3 normalIn = getNormalAt(worldIn);
        Vector3 normalOut = getNormalAt(worldOut);
        Intersection in = new Intersection(worldIn, normalIn, tIn, this);
        Intersection out = new Intersection(worldOut, normalOut, tOut, this);

        intervals.add(new IntersectionInterval(tIn, tOut, in, out));
        tIn = -1; // Reset for next interval
    }

    // Move forward
    t += Math.max(dist * 0.5, stepSize);
}

```

```

    if (t > maxDistance) {
        // Handle case where ray ends inside (optional)
        break;
    }
}

return intervals;
}

private double signedDistanceFunction(Point3 p) {
    double theta = Math.atan2(p.y, p.x);
    double phi = (this.q * theta) / this.p; // Düzeltme: this.p kullanıldı

    Point3 knotPos = new Point3(
        (R + r * Math.cos(phi)) * Math.cos(theta),
        (R + r * Math.cos(phi)) * Math.sin(theta),
        r * Math.sin(phi)
    );

    return p.subtract(knotPos).length() - (r * 0.3);
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    Point3 localPoint = inverseTransform.transformPoint(worldPoint);

    double eps = 0.001;
    double dx = signedDistanceFunction(localPoint.add(new Vector3(eps,
0, 0))) -
        signedDistanceFunction(localPoint.add(new Vector3(-eps, 0, 0)));
    double dy = signedDistanceFunction(localPoint.add(new Vector3(0,
eps, 0))) -
        signedDistanceFunction(localPoint.add(new Vector3(0, -eps, 0)));
    double dz = signedDistanceFunction(localPoint.add(new Vector3(0, 0,
eps))) -
        signedDistanceFunction(localPoint.add(new Vector3(0, 0, -eps)));

    Vector3 localNormal = new Vector3(dx, dy, dz).normalize();
    return

```

```
inverseTransform.inverseTransposeForNormal().transformVector(localNormal).normalize();
}

@Override
public void setTransform(Matrix4 transform) {
    this.transform = new Matrix4(transform);
    this.inverseTransform = transform.inverse();
}

@Override
public Matrix4 getTransform() {
    return new Matrix4(transform);
}

@Override
public Matrix4 getInverseTransform() {
    return new Matrix4(inverseTransform);
}

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

/**
 * Usage examples:
 *
 * // Classic trefoil knot (2,3)
 * TorusKnot trefoil = new TorusKnot(1.5, 0.4, 2, 3);
 * trefoil.setMaterial(new GlossyMaterial(Color.CYAN, 0.3));
 * trefoil.setTransform(Matrix4.rotationX(Math.PI/2));
 *
 * // More complex (3,5) knot
```

```

* TorusKnot complexKnot = new TorusKnot(2.0, 0.3, 3, 5);
* complexKnot.setTransform(
*     Matrix4.translate(0, 1, 0)
*         .multiply(Matrix4.scale(1.2, 1.2, 1.2)))
* );
*/
// =====
// File: /net/elenamurat/shape/Plane.java
// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/**
 * Plane class represents an infinite plane in 3D space.
 * Implements EMShape interface, handling transformations internally.
 */
public class Plane implements EMShape {
    private Material material;

    private final Point3 pointOnPlane;
    private final Vector3 normal;

    private Matrix4 transform;
    private Matrix4 inverseTransform; // Inverse transform'u burada tutacağınız

    public Plane(Point3 pointOnPlane, Vector3 normal) {
        this.pointOnPlane = pointOnPlane;
        this.normal = normal.normalize();

        this.transform = new Matrix4();
        this.inverseTransform = new Matrix4(); // Başlangıçta identity olarak
ayarla

```

```
}
```

```
@Override  
public void setMaterial(Material material) {  
    this.material = material;  
}
```

```
@Override  
public Material getMaterial() {  
    return this.material;  
}
```

```
@Override  
public void setTransform(Matrix4 transform) {  
    this.transform = transform;  
    this.inverseTransform = transform.inverse(); // setTransform  
    çağrıldığında tersini hesapla  
    if(this.inverseTransform == null) {  
        System.err.println("Warning: Plane's transform is non-invertible.  
Inverse transform set to null.");  
    }  
}
```

```
@Override  
public Matrix4 getTransform() {  
    return this.transform;  
}
```

```
@Override // EMShape'te tanımlı olduğu için @Override  
public Matrix4 getInverseTransform() {  
    return this.inverseTransform; // Saklanan ters dönüşümü döndür  
}
```

```
@Override  
public double intersect(Ray ray) {  
    // Artık yerel olarak hesaplamıyoruz, saklanan inverseTransform'u  
    kullanıyoruz  
    if(this.inverseTransform == null) {  
        System.err.println("Error: Plane's inverse transform is null during
```

```

intersect. Returning no intersection.");
    return -1;
}

Point3 localOrigin =
this.inverseTransform.transformPoint(ray.getOrigin());
Vector3 localDirection =
this.inverseTransform.transformVector(ray.getDirection()).normalize();
Ray localRay = new Ray(localOrigin, localDirection);

double denom = localRay.getDirection().dot(this.normal);

if (Math.abs(denom) < Ray.EPSILON) {
    double num =
this.pointOnPlane.subtract(localRay.getOrigin()).dot(this.normal);
    if (Math.abs(num) < Ray.EPSILON) {
        return -1;
    }
    return -1;
}

double t =
this.pointOnPlane.subtract(localRay.getOrigin()).dot(this.normal) / denom;

if (t > Ray.EPSILON) {
    return t;
}
return -1;
}

/**
 * Calculates all intersection intervals between a ray and this infinite
plane.
 * Since a plane is infinitely thin, the entry and exit points are considered
the same.
 * @param ray The ray to test, in world coordinates.
 * @return A list containing a single degenerate interval if intersected,
otherwise empty list.
*/

```

```

@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into the plane's local space
    if (this.inverseTransform == null) {
        return java.util.Collections.emptyList();
    }

    Point3 localOrigin =
    this.inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
    this.inverseTransform.transformVector(ray.getDirection()).normalize();
    Ray localRay = new Ray(localOrigin, localDirection);

    double denom = localDirection.dot(this.normal);

    // Parallel to the plane
    if (Math.abs(denom) < Ray.EPSILON) {
        return java.util.Collections.emptyList();
    }

    double t = this.pointOnPlane.subtract(localOrigin).dot(this.normal) /
denom;

    if (t <= Ray.EPSILON) {
        return java.util.Collections.emptyList();
    }

    // Create a single degenerate interval (in and out at the same point)
    Point3 hitPoint = ray.pointAtParameter(t);
    Vector3 hitNormal = getNormalAt(hitPoint);
    Intersection hit = new Intersection(hitPoint, hitNormal, t, this);

    IntersectionInterval interval = IntersectionInterval.point(t, hit);
    return java.util.Arrays.asList(interval);
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // Artık yerel olarak hesaplamıyoruz, saklanan inverseTransform'u

```

kullanıyoruz

```
    if (this.inverseTransform == null) {
        System.err.println("Error: Plane's inverse transform is null during
getNormalAt. Returning default normal.");
        return new Vector3(0, 1, 0);
    }
    Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal();
    if (normalTransformMatrix == null) {
        System.err.println("Error: Plane's normal transform matrix is null.
Returning default normal.");
        return new Vector3(0, 1, 0);
    }
    return
normalTransformMatrix.transformVector(this.normal).normalize();
}
```

```
// =====
// File: /net/elenamurat/shape/DifferenceCSG.java
// =====
```

```
package net.elena.murat.shape;
```

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import net.elena.murat.math.*;
```

```
/**
```

```
 * Represents a Constructive Solid Geometry (CSG) Difference operation.
 * The resulting shape is the difference of two shapes: A - B.
 * A point is inside the difference if it is inside shape A and outside shape
B.
 */
```

```
public class DifferenceCSG extends CSGShape {
```

```

/**
 * Constructs a DifferenceCSG from two shapes.
 * @param left The first shape (left operand, the one being subtracted
from).
 * @param right The second shape (right operand, the one being
subtracted).
 */
public DifferenceCSG(EMShape left, EMShape right) {
    super(left, right);
}

/**
 * Combines the intersection intervals of two shapes using the Difference
operation.
 * The difference is formed by finding intervals where the ray is inside
'left' and outside 'right'.
 * @param a List of intervals from the left shape (A).
 * @param b List of intervals from the right shape (B).
 * @return The resulting list of intervals for the difference (A - B).
*/
@Override
protected List<IntersectionInterval> combine(
    List<IntersectionInterval> a,
    List<IntersectionInterval> b) {

    // 1. If A has no intersections, result is empty
    if (a.isEmpty()) {
        return Collections.emptyList();
    }

    // 2. If B has no intersections, result is just A
    if (b.isEmpty()) {
        return new ArrayList<>(a);
    }

    // 3. Sort intervals by tIn
    List<IntersectionInterval> sortedA = new ArrayList<>(a);
    List<IntersectionInterval> sortedB = new ArrayList<>(b);
    Collections.sort(sortedA, (ia, ib) -> Double.compare(ia.tIn, ib.tIn));
}

```

```

Collections.sort(sortedB, (ia, ib) -> Double.compare(ia.tIn, ib.tIn));

List<IntersectionInterval> result = new ArrayList<>();

for (IntersectionInterval intervalA : sortedA) {
    double currentTIn = intervalA.tIn;
    double currentTOut = intervalA.tOut;

    // For each interval in A, subtract all overlaps with B
    for (IntersectionInterval intervalB : sortedB) {
        // If B interval starts after A ends, no overlap
        if (intervalB.tIn >= currentTOut - Ray.EPSILON) {
            break;
        }

        // If B interval ends before A starts, no overlap
        if (intervalB.tOut <= currentTIn + Ray.EPSILON) {
            continue;
        }

        // There is an overlap
        double overlapTIn = Math.max(currentTIn, intervalB.tIn);
        double overlapTOut = Math.min(currentTOut, intervalB.tOut);

        // Add part before overlap (if exists)
        if (currentTIn < overlapTIn - Ray.EPSILON) {
            result.add(createInterval(intervalA, currentTIn, overlapTIn));
        }

        // Update current interval start
        currentTIn = overlapTOut;
    }

    // Add remaining part after all B intervals
    if (currentTIn < currentTOut - Ray.EPSILON) {
        result.add(createInterval(intervalA, currentTIn, currentTOut));
    }
}

```

```

        return result;
    }

/***
 * Helper method to create a new IntersectionInterval with correct hit
data.
 * Uses linear interpolation to estimate point and normal at new t values.
 * @param original The original interval to copy data from.
 * @param tIn New tIn value.
 * @param tOut New tOut value.
 * @return A new IntersectionInterval.
*/
private IntersectionInterval createInterval(IntersectionInterval original,
double tIn, double tOut) {
    // Interpolate points
    Point3 pointIn = original.in.getPoint().add(
        original.out.getPoint().subtract(original.in.getPoint())
        .scale((tIn - original.tIn) / (original.tOut - original.tIn +
Ray.EPSILON))
    );
    Point3 pointOut = original.in.getPoint().add(
        original.out.getPoint().subtract(original.in.getPoint())
        .scale((tOut - original.tIn) / (original.tOut - original.tIn +
Ray.EPSILON))
    );

    // Use original normals (approximation)
    Vector3 normalIn = original.in.getNormal();
    Vector3 normalOut = original.out.getNormal();

    Intersection in = new Intersection(pointIn, normalIn, tIn, this);
    Intersection out = new Intersection(pointOut, normalOut, tOut, this);

    return new IntersectionInterval(tIn, tOut, in, out);
}

```

```

// =====
// File: /net/elenamurat/shape/UnionCSG.java
// =====

package net.elena.murat.shape;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

import net.elena.murat.math.*;

/**
 * Represents a Constructive Solid Geometry (CSG) Union operation.
 * The resulting shape is the union of two shapes: A ∪ B.
 * A point is inside the union if it is inside shape A, shape B, or both.
 */
public class UnionCSG extends CSGShape {

    /**
     * Constructs a UnionCSG from two shapes.
     * @param left The first shape (left operand).
     * @param right The second shape (right operand).
     */
    public UnionCSG(EMShape left, EMShape right) {
        super(left, right);
    }

    /**
     * Combines the intersection intervals of two shapes using the Union
     * operation.
     * The union is formed by merging overlapping intervals from both
     * shapes.
     * @param a List of intervals from the left shape.
     * @param b List of intervals from the right shape.
     * @return The resulting list of non-overlapping intervals for the union.
     */
    @Override
    protected List<IntersectionInterval> combine(

```

```

List<IntersectionInterval> a,
List<IntersectionInterval> b) {

    // 1. Combine all intervals from both shapes
    List<IntersectionInterval> all = new ArrayList<>();
    all.addAll(a);
    all.addAll(b);

    // 2. Sort intervals by tIn (entry point)
    Collections.sort(all, (ia, ib) -> Double.compare(ia.tIn, ib.tIn));

    // 3. Merge overlapping intervals
    List<IntersectionInterval> merged = new ArrayList<>();
    if (all.isEmpty()) return merged;

    IntersectionInterval current = all.get(0);

    for (int i = 1; i < all.size(); i++) {
        IntersectionInterval next = all.get(i);

        // If current interval overlaps or touches the next one
        if (current.tOut >= next.tIn - Ray.EPSILON) {
            // Extend the current interval's tOut
            double newTOut = Math.max(current.tOut, next.tOut);
            current = new IntersectionInterval(
                current.tIn,
                newTOut,
                current.in, // Keep original in hit
                next.out // Use new out hit
            );
        } else {
            // No overlap, finalize current interval
            merged.add(current);
            current = next;
        }
    }

    // Add the last interval
    merged.add(current);
}

```

```

        return merged;
    }

}

// =====
// File: /net/elenamurat/shape/Cube.java
// =====

package net.elena.murat.shape;

import java.util.List;

// Custom imports
import net.elena.murat.material.Material;
import net.elena.murat.math.*;

/**
 * Represents a cube in the scene.
 * It can be defined by a side length (centered at origin) or by two corner
points.
 * It implements EMShape to support transformations and materials.
 */
public class Cube implements EMShape {

    private Point3 minBounds; // Minimum corner of the cube in local space
    private Point3 maxBounds; // Maximum corner of the cube in local space
    private Material material;
    private Matrix4 transform;
    private Matrix4 inverseTransform;
    private Matrix4 inverseTransposeTransformForNormal; // For correct
normal transformation

    /**
     * Constructs a Cube with the given side length, centered at the origin in
its local space.
     * @param sideLength The length of each side of the cube.
     */

```

```

public Cube(double sideLength) {
    double halfSide = sideLength / 2.0;
    this.minBounds = new Point3(-halfSide, -halfSide, -halfSide);
    this.maxBounds = new Point3(halfSide, halfSide, halfSide);
    this.transform = Matrix4.identity(); // Initialize with identity
    updateTransforms(); // Initial calculation of inverse transforms
}

/**
 * Constructs a Cube from two corner points in its local space.
 * The cube will be axis-aligned within its local coordinate system.
 * @param min The minimum corner of the cube (e.g., lower-left-back).
 * @param max The maximum corner of the cube (e.g., upper-right-front).
 */
public Cube(Point3 min, Point3 max) {
    this.minBounds = min;
    this.maxBounds = max;
    this.transform = Matrix4.identity(); // Initialize with identity
    updateTransforms(); // Initial calculation of inverse transforms
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setTransform(Matrix4 transform) {
    // Create a deep copy of the incoming matrix to prevent external
    modifications
    this.transform = new Matrix4(transform);
    updateTransforms(); // Update inverse matrices when transform changes
}

```

```
@Override
public Matrix4 getTransform() {
    return transform;
}

@Override
public Matrix4 getInverseTransform() {
    return inverseTransform;
}

/***
 * Updates the inverse and inverse transpose transforms whenever the
main transform changes.
 * This method is called internally by setTransform and the constructor.
 */
private void updateTransforms() {
    this.inverseTransform = this.transform.inverse();
    this.inverseTransposeTransformForNormal =
this.transform.inverseTransposeForNormal();
}

/***
 * Calculates the intersection of a ray with the cube.
 * This method transforms the ray into the object's local space,
 * performs the intersection test, and returns the 't' value.
 *
 * @param ray The ray to intersect with the cube (in world coordinates).
 * @return The distance 't' along the ray to the closest intersection point,
 * or Double.POSITIVE_INFINITY if no intersection.
 */
@Override
public double intersect(Ray ray) {
    // Ensure transforms are not null before proceeding
    if (inverseTransform == null || inverseTransposeTransformForNormal
== null) {
        System.err.println("Error: Cube transforms are null. Cannot
intersect.");
        return Double.POSITIVE_INFINITY; // Return infinity if transforms
```

are invalid

}

// 1. Transform the ray into the cube's local space

Ray localRay = new Ray(

inverseTransform.transformPoint(ray.getOrigin()),

inverseTransform.transformVector(ray.getDirection()).normalize() //

Normalize direction after transformation

);

double tMin = Double.NEGATIVE_INFINITY;

double tMax = Double.POSITIVE_INFINITY;

// Kesişim hesaplamaları için minBounds ve maxBounds kullanılıyor

// X-düzlemleriyle kesişim

if (Math.abs(localRay.getDirection().x) < Ray.EPSILON) {

if (localRay.getOrigin().x < minBounds.x || localRay.getOrigin().x > maxBounds.x) {

return Double.POSITIVE_INFINITY;

}

} else {

double t1 = (minBounds.x - localRay.getOrigin().x) /

localRay.getDirection().x;

double t2 = (maxBounds.x - localRay.getOrigin().x) /

localRay.getDirection().x;

if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }

tMin = Math.max(tMin, t1);

tMax = Math.min(tMax, t2);

if (tMin > tMax) return Double.POSITIVE_INFINITY;

}

// Y-düzlemleriyle kesişim

if (Math.abs(localRay.getDirection().y) < Ray.EPSILON) {

if (localRay.getOrigin().y < minBounds.y || localRay.getOrigin().y > maxBounds.y) {

return Double.POSITIVE_INFINITY;

}

} else {

double t1 = (minBounds.y - localRay.getOrigin().y) /

```

localRay.getDirection().y;
    double t2 = (maxBounds.y - localRay.getOrigin().y) /
localRay.getDirection().y;
    if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return Double.POSITIVE_INFINITY;
    }

// Z-düzlemleriyle kesişim
if (Math.abs(localRay.getDirection().z) < Ray.EPSILON) {
    if (localRay.getOrigin().z < minBounds.z || localRay.getOrigin().z >
maxBounds.z) {
        return Double.POSITIVE_INFINITY;
    }
} else {
    double t1 = (minBounds.z - localRay.getOrigin().z) /
localRay.getDirection().z;
    double t2 = (maxBounds.z - localRay.getOrigin().z) /
localRay.getDirection().z;
    if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return Double.POSITIVE_INFINITY;
    }

double t = tMin;
if (t < Ray.EPSILON) { // If closest intersection is behind or too close to
origin
    t = tMax; // Try the farther intersection
    if (t < Ray.EPSILON) { // If farther intersection is also behind
        return Double.POSITIVE_INFINITY; // No valid intersection
    }
}

return t; // Return the valid intersection distance
}

/***

```

```

* Calculates all intersection intervals between a ray and this cube.
* Uses the slab method to find entry and exit points on the six faces.
* @param ray The ray to test, in world coordinates.
* @return A list of IntersectionInterval objects. Empty if no intersection.
*/
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Check for valid transforms
    if (inverseTransform == null || inverseTransposeTransformForNormal
        == null) {
        return java.util.Collections.emptyList();
    }

    // 2. Transform the ray into local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize()
    );

    double tMin = Double.NEGATIVE_INFINITY;
    double tMax = Double.POSITIVE_INFINITY;

    // Slab intersection on X, Y, Z axes
    double[][] slabs = {
        { minBounds.x, maxBounds.x, localRay.getDirection().x,
            localRay.getOrigin().x },
        { minBounds.y, maxBounds.y, localRay.getDirection().y,
            localRay.getOrigin().y },
        { minBounds.z, maxBounds.z, localRay.getDirection().z,
            localRay.getOrigin().z }
    };

    for (double[] slab : slabs) {
        double minBound = slab[0], maxBound = slab[1];
        double dir = slab[2], origin = slab[3];

        if (Math.abs(dir) < Ray.EPSILON) {
            if (origin < minBound || origin > maxBound) {
                return java.util.Collections.emptyList();
            }
        }
    }
}

```

```

        }
    } else {
        double t1 = (minBound - origin) / dir;
        double t2 = (maxBound - origin) / dir;
        if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return java.util.Collections.emptyList();
    }
}

// 3. Now we have tMin (entry) and tMax (exit)
if (tMax < Ray.EPSILON) return java.util.Collections.emptyList();
if (tMin < Ray.EPSILON) tMin = tMax; // Ray is inside the cube

// 4. Create Intersection objects
Point3 pointIn = ray.pointAtParameter(tMin);
Vector3 normalIn = getNormalAt(pointIn);
Intersection in = new Intersection(pointIn, normalIn, tMin, this);

Point3 pointOut = ray.pointAtParameter(tMax);
Vector3 normalOut = getNormalAt(pointOut);
Intersection out = new Intersection(pointOut, normalOut, tMax, this);

// 5. Return the interval
return java.util.Arrays.asList(new IntersectionInterval(tMin, tMax, in,
out));
}

/***
 * Returns the surface normal at a given point on the cube's surface.
 * The point is in world coordinates. The normal is calculated in local
space
 * and then transformed back to world space.
 *
 * @param worldPoint The point on the cube's surface in world
coordinates.
 * @return The normalized normal vector at that point in world
coordinates.
 */

```

```

*/
@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // Ensure transforms are not null before proceeding
    if (inverseTransform == null || inverseTransposeTransformForNormal
        == null) {
        System.err.println("Error: Cube transforms are null. Cannot compute
normal.");
        // Fallback: return a default normal or throw an exception
        return new Vector3(0, 1, 0);
    }

    // Transform the world point to the cube's local space
    Point3 localPoint =
this.getInverseTransform().transformPoint(worldPoint);
    Vector3 localNormal = null;

    // Use a slightly larger epsilon for normal calculation to avoid ambiguity
    at edges/corners
    //double normalEpsilon = Ray.EPSILON * 10;
    double normalEpsilon = 1e-3;

    // Determine which face was hit to get the normal (based on min/max
    bounds)
    if (Math.abs(localPoint.x - maxBounds.x) < normalEpsilon) {
        localNormal = new Vector3(1, 0, 0);
    } else if (Math.abs(localPoint.x - minBounds.x) < normalEpsilon) {
        localNormal = new Vector3(-1, 0, 0);
    } else if (Math.abs(localPoint.y - maxBounds.y) < normalEpsilon) {
        localNormal = new Vector3(0, 1, 0);
    } else if (Math.abs(localPoint.y - minBounds.y) < normalEpsilon) {
        localNormal = new Vector3(0, -1, 0);
    } else if (Math.abs(localPoint.z - maxBounds.z) < normalEpsilon) {
        localNormal = new Vector3(0, 0, 1);
    } else if (Math.abs(localPoint.z - minBounds.z) < normalEpsilon) {
        localNormal = new Vector3(0, 0, -1);
    } else {
        // Fallback for floating point inaccuracies near edges/corners.
        // This attempts to find the closest face based on the local point's
    }
}

```

coordinates.

```
    double[] dists = {
        Math.abs(localPoint.x - maxBounds.x),
        Math.abs(localPoint.x - minBounds.x),
        Math.abs(localPoint.y - maxBounds.y),
        Math.abs(localPoint.y - minBounds.y),
        Math.abs(localPoint.z - maxBounds.z),
        Math.abs(localPoint.z - minBounds.z)
    };
    int minIdx = 0;
    for(int i = 1; i < 6; i++) {
        if (dists[i] < dists[minIdx]) {
            minIdx = i;
        }
    }
    switch(minIdx) {
        case 0: localNormal = new Vector3(1, 0, 0); break;
        case 1: localNormal = new Vector3(-1, 0, 0); break;
        case 2: localNormal = new Vector3(0, 1, 0); break;
        case 3: localNormal = new Vector3(0, -1, 0); break;
        case 4: localNormal = new Vector3(0, 0, 1); break;
        case 5: localNormal = new Vector3(0, 0, -1); break;
        default: localNormal = new Vector3(0, 1, 0); // Should not happen
    }
    //System.err.println("Warning: Cube normal fallback used due to
floating point inaccuracy.");
}

// Transform the local normal back to world space
// Use inverse transpose for correct normal transformation
return
this.inverseTransposeTransformForNormal.transformVector(localNormal)
.normalize();
}

// =====
// File: /net/elena/murat/shape/Hyperboloid.java
```

```

// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/**
 * Single-sheet hyperboloid shape ( $x^2/a^2 + y^2/b^2 - z^2/c^2 = 1$ )
 * Fully implements the EMShape interface
 */
public class Hyperboloid implements EMShape {
    private final double a, b, c; // Hyperboloid parameters
    private final double height; // Height limit along Z-axis ( $|z| \leq height$ )
    private Material material;
    private Matrix4 transform = Matrix4.identity();
    private Matrix4 inverseTransform = Matrix4.identity();

    /**
     * Standard hyperboloid constructor (a=1, b=1, c=1, height=5)
     */
    public Hyperboloid() {
        this(1.0, 1.0, 1.0, 5.0);
    }

    /**
     * Parameterized hyperboloid constructor
     * @param a X-axis parameter
     * @param b Y-axis parameter
     * @param c Z-axis parameter
     * @param height Maximum height (taken as absolute value)
     */
    public Hyperboloid(double a, double b, double c, double height) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.height = Math.abs(height);
    }
}

```

```
}
```

```
@Override
public double intersect(Ray ray) {
    // Transform ray to local coordinate system
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
inverseTransform.transformVector(ray.getDirection()).normalize();
    Ray localRay = new Ray(localOrigin, localDirection);

    // Calculate quadratic equation coefficients: At2 + Bt + C = 0
    double ox = localRay.getOrigin().x;
    double oy = localRay.getOrigin().y;
    double oz = localRay.getOrigin().z;
    double dx = localRay.getDirection().x;
    double dy = localRay.getDirection().y;
    double dz = localRay.getDirection().z;

    double a2 = a * a;
    double b2 = b * b;
    double c2 = c * c;

    double A = (dx*dx)/a2 + (dy*dy)/b2 - (dz*dz)/c2;
    double B = 2.0 * ((ox*dx)/a2 + (oy*dy)/b2 - (oz*dz)/c2);
    double C = (ox*ox)/a2 + (oy*oy)/b2 - (oz*oz)/c2 - 1.0;

    double discriminant = B*B - 4.0*A*C;

    if (discriminant < 0.0) {
        return -1.0; // No intersection
    }

    double sqrtDiscriminant = Math.sqrt(discriminant);
    double t1 = (-B - sqrtDiscriminant) / (2.0*A);
    double t2 = (-B + sqrtDiscriminant) / (2.0*A);

    // Check valid intersection points
    Point3 p1 = localRay.pointAtParameter(t1);
    Point3 p2 = localRay.pointAtParameter(t2);
```

```

boolean validT1 = isValidIntersection(p1);
boolean validT2 = isValidIntersection(p2);

if (validT1 && validT2) {
    return Math.min(t1, t2);
} else if (validT1) {
    return t1;
} else if (validT2) {
    return t2;
}

return -1.0;
}

/***
 * Calculates all intersection intervals between a ray and this hyperboloid.
 * The ray is transformed into the hyperboloid's local space for
calculation.
 * The method solves the quadratic equation for the hyperboloid and
returns
 * intervals where the ray is inside the shape (between valid
intersections).
 * @param ray The ray to test, in world coordinates.
 * @return A list of IntersectionInterval objects. Empty if no valid
intersection.
*/
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into local space
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
inverseTransform.transformVector(ray.getDirection()).normalize();
    Ray localRay = new Ray(localOrigin, localDirection);

    // 2. Coefficients for quadratic equation: At2 + Bt + C = 0
    double ox = localOrigin.x;
    double oy = localOrigin.y;
    double oz = localOrigin.z;
}

```

```

double dx = localDirection.x;
double dy = localDirection.y;
double dz = localDirection.z;

double a2 = a * a;
double b2 = b * b;
double c2 = c * c;

double A = (dx*dx)/a2 + (dy*dy)/b2 - (dz*dz)/c2;
double B = 2.0 * ((ox*dx)/a2 + (oy*dy)/b2 - (oz*dz)/c2);
double C = (ox*ox)/a2 + (oy*oy)/b2 - (oz*oz)/c2 - 1.0;

double discriminant = B*B - 4.0*A*C;
if (discriminant < 0.0) {
    return java.util.Collections.emptyList();
}

double sqrtDiscriminant = Math.sqrt(discriminant);
double t1 = (-B - sqrtDiscriminant) / (2.0 * A);
double t2 = (-B + sqrtDiscriminant) / (2.0 * A);

// Ensure t1 is entry, t2 is exit
if (t1 > t2) {
    double temp = t1;
    t1 = t2;
    t2 = temp;
}

// 3. Check validity (within height bounds)
Point3 p1 = localRay.pointAtParameter(t1);
Point3 p2 = localRay.pointAtParameter(t2);

boolean validT1 = isValidIntersection(p1);
boolean validT2 = isValidIntersection(p2);

List<IntersectionInterval> intervals = new java.util.ArrayList<>();

if (validT1 && validT2) {
    // Both intersections valid → full interval
}

```

```

Point3 worldIn = ray.pointAtParameter(t1);
Point3 worldOut = ray.pointAtParameter(t2);
Vector3 normalIn = getNormalAt(worldIn);
Vector3 normalOut = getNormalAt(worldOut);
Intersection in = new Intersection(worldIn, normalIn, t1, this);
Intersection out = new Intersection(worldOut, normalOut, t2, this);
intervals.add(new IntersectionInterval(t1, t2, in, out));
} else if (validT1) {
    // Only t1 valid → degenerate interval (e.g., ray ends inside)
    Point3 worldIn = ray.pointAtParameter(t1);
    Vector3 normalIn = getNormalAt(worldIn);
    Intersection hit = new Intersection(worldIn, normalIn, t1, this);
    intervals.add(IntersectionInterval.point(t1, hit));
} else if (validT2) {
    // Only t2 valid → degenerate interval
    Point3 worldIn = ray.pointAtParameter(t2);
    Vector3 normalIn = getNormalAt(worldIn);
    Intersection hit = new Intersection(worldIn, normalIn, t2, this);
    intervals.add(IntersectionInterval.point(t2, hit));
}
return intervals;
}

```

```

private boolean isValidIntersection(Point3 localPoint) {
    // Height boundary check
    return Math.abs(localPoint.z) <= height;
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    Point3 localPoint = inverseTransform.transformPoint(worldPoint);

    // Hyperboloid surface normal (gradient)
    Vector3 localNormal = new Vector3(
        2.0 * localPoint.x / (a * a),
        2.0 * localPoint.y / (b * b),
        -2.0 * localPoint.z / (c * c)
    ).normalize();
}

```

```
// Transform normal to world coordinates (using inverse transpose)
return
inverseTransform.inverseTransposeForNormal().transformVector(localNormal).normalize();
}

@Override
public void setTransform(Matrix4 transform) {
    this.transform = new Matrix4(transform);
    this.inverseTransform = transform.inverse();
}

@Override
public Matrix4 getTransform() {
    return new Matrix4(transform);
}

@Override
public Matrix4 getInverseTransform() {
    return new Matrix4(inverseTransform);
}

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

// Helper getter methods
public double getA() { return a; }
public double getB() { return b; }
public double getC() { return c; }
public double getHeight() { return height; }
}
```

```

/**
 * Usage examples:
 *
 * // Standard hyperboloid
 * Hyperboloid hyperboloid = new Hyperboloid();
 * hyperboloid.setMaterial(new GlossyMaterial(Color.RED, 0.3));
 *
 * // Customized hyperboloid
 * Hyperboloid customHyper = new Hyperboloid(1.5, 0.8, 1.2, 3.0);
 * customHyper.setTransform(Matrix4.rotationY(Math.PI/4));
 *
 * // Transformed hyperboloid
 * Hyperboloid transformed = new Hyperboloid();
 * transformed.setTransform(
 *     Matrix4.translate(2, 0, 0)
 *     .multiply(Matrix4.scale(1, 2, 1)) // Stretch along Y-axis
 * );
 */

```

```

// =====
// File: /net/elenamurat/shape/Box.java
// =====

```

```

package net.elena.murat.shape;

import java.util.List;

// Custom imports
import net.elena.murat.material.Material;
import net.elena.murat.math.*;

/**

 * Represents an axis-aligned rectangular prism (or cuboid) in 3D space.
 * It implements EMShape to support transformations and materials.
 * The prism is defined by its width, height, and depth, centered at its local
origin (0,0,0).
*/

```

```

public class Box implements EMShape {

    private final double width;
    private final double height;
    private final double depth;

    // EMShape interface transformation matrices and material
    private Matrix4 transform;
    private Matrix4 inverseTransform;
    private Matrix4 inverseTransposeTransformForNormal; // For correct
normal transformation
    private Material material;

    /**
     * Constructs a Box with specified dimensions.
     * The prism is initially axis-aligned and centered at (0,0,0) in its local
space.
     *
     * @param width The width of the prism along the local X-axis.
     * @param height The height of the prism along the local Y-axis.
     * @param depth The depth of the prism along the local Z-axis.
     */
    public Box(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
        // Default transform is identity.
        this.transform = Matrix4.identity();
        updateTransforms(); // Inverse transforms are calculated initially
    }

    /**
     * Constructs a Box with specified dimensions and a material.
     *
     * @param width The width of the prism along the local X-axis.
     * @param height The height of the prism along the local Y-axis.
     * @param depth The depth of the prism along the local Z-axis.
     * @param material The material applied to the prism.
     */
}

```

```
public Box(double width, double height, double depth, Material material)
{
    this(width, height, depth);
    this.setMaterial(material);
}
```

```
// --- EMShape Interface Methods Implementation ---
```

```
@Override
```

```
public void setTransform(Matrix4 transform) {
    // Create a deep copy of the incoming matrix to prevent external
    modifications
    this.transform = new Matrix4(transform);
    updateTransforms(); // Update inverse matrices when transform changes
}
```

```
@Override
```

```
public Matrix4 getTransform() {
    return this.transform;
}
```

```
@Override
```

```
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}
```

```
/**
```

```
* Updates the inverse and inverse transpose transforms whenever the
main transform changes.
```

```
* This method is called internally by setTransform and the constructor.
*/
```

```
private void updateTransforms() {
    this.inverseTransform = this.transform.inverse();
    this.inverseTransposeTransformForNormal =
this.transform.inverseTransposeForNormal();
}
```

```
@Override
```

```
public void setMaterial(Material material) {
```

```

    this.material = material;
}

@Override
public Material getMaterial() {
    return this.material;
}

/***
 * Calculates the intersection of a ray with the rectangular prism.
 * This method transforms the ray into the object's local space,
 * performs the intersection test, and returns the 't' value.
 *
 * @param ray The ray to intersect with the prism (in world coordinates).
 * @return The distance 't' along the ray to the closest intersection point,
 * or Double.POSITIVE_INFINITY if no intersection.
 */
@Override
public double intersect(Ray ray) {
    // Ensure transforms are not null before proceeding
    if (inverseTransform == null || inverseTransposeTransformForNormal
    == null) {
        System.err.println("Error: Box transforms are null. Cannot intersect.");
        return Double.POSITIVE_INFINITY; // Return infinity if transforms
    are invalid
    }

    // 1. Transform the ray into the prism's local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize() //
    Normalize direction after transformation
    );

    double tMin = Double.NEGATIVE_INFINITY;
    double tMax = Double.POSITIVE_INFINITY;

    // Calculate half dimensions for easier calculations
    double halfWidth = width / 2.0;

```

```

double halfHeight = height / 2.0;
double halfDepth = depth / 2.0;

// Intersection with X-planes (slab method)
// Handle cases where ray direction component is zero to avoid division
by zero
if (Math.abs(localRay.getDirection().x) < Ray.EPSILON) {
    if (localRay.getOrigin().x < -halfWidth || localRay.getOrigin().x >
halfWidth) {
        return Double.POSITIVE_INFINITY; // Ray is parallel and outside
the slab
    }
} else {
    double t1 = (-halfWidth - localRay.getOrigin().x) /
localRay.getDirection().x;
    double t2 = (halfWidth - localRay.getOrigin().x) /
localRay.getDirection().x;
    if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; } // Ensure t1 is min,
t2 is max
    tMin = Math.max(tMin, t1);
    tMax = Math.min(tMax, t2);
    if (tMin > tMax) return Double.POSITIVE_INFINITY; // No
intersection
}

// Intersection with Y-planes (slab method)
if (Math.abs(localRay.getDirection().y) < Ray.EPSILON) {
    if (localRay.getOrigin().y < -halfHeight || localRay.getOrigin().y >
halfHeight) {
        return Double.POSITIVE_INFINITY;
    }
} else {
    double t1 = (-halfHeight - localRay.getOrigin().y) /
localRay.getDirection().y;
    double t2 = (halfHeight - localRay.getOrigin().y) /
localRay.getDirection().y;
    if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
    tMin = Math.max(tMin, t1);
    tMax = Math.min(tMax, t2);
}

```

```

    if (tMin > tMax) return Double.POSITIVE_INFINITY;
}

// Intersection with Z-planes (slab method)
if (Math.abs(localRay.getDirection().z) < Ray.EPSILON) {
    if (localRay.getOrigin().z < -halfDepth || localRay.getOrigin().z >
halfDepth) {
        return Double.POSITIVE_INFINITY;
    }
} else {
    double t1 = (-halfDepth - localRay.getOrigin().z) /
localRay.getDirection().z;
    double t2 = (halfDepth - localRay.getOrigin().z) /
localRay.getDirection().z;
    if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
    tMin = Math.max(tMin, t1);
    tMax = Math.min(tMax, t2);
    if (tMin > tMax) return Double.POSITIVE_INFINITY;
}

double t = tMin;
if (t < Ray.EPSILON) { // If closest intersection is behind or too close to
origin
    t = tMax; // Try the farther intersection
    if (t < Ray.EPSILON) { // If farther intersection is also behind
        return Double.POSITIVE_INFINITY; // No valid intersection
    }
}

return t; // Return the valid intersection distance
}

/**
 * Calculates all intersection intervals between a ray and this rectangular
prism.
 * Uses the slab method to find entry and exit points on the six faces.
 * @param ray The ray to test, in world coordinates.
 * @return A list of IntersectionInterval objects. Empty if no intersection.
*/

```

```

@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Check for valid transforms
    if (inverseTransform == null || inverseTransposeTransformForNormal
        == null) {
        return java.util.Collections.emptyList();
    }

    // 2. Transform the ray into local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize()
    );

    double tMin = Double.NEGATIVE_INFINITY;
    double tMax = Double.POSITIVE_INFINITY;

    double halfWidth = width / 2.0;
    double halfHeight = height / 2.0;
    double halfDepth = depth / 2.0;

    // Slab intersection on X, Y, Z axes
    double[][] slabs = {
        { -halfWidth, halfWidth, localRay.getDirection().x,
        localRay.getOrigin().x },
        { -halfHeight, halfHeight, localRay.getDirection().y,
        localRay.getOrigin().y },
        { -halfDepth, halfDepth, localRay.getDirection().z,
        localRay.getOrigin().z }
    };

    for (double[] slab : slabs) {
        double minBound = slab[0], maxBound = slab[1];
        double dir = slab[2], origin = slab[3];

        if (Math.abs(dir) < Ray.EPSILON) {
            if (origin < minBound || origin > maxBound) {
                return java.util.Collections.emptyList();
            }
        }
    }
}

```

```

    } else {
        double t1 = (minBound - origin) / dir;
        double t2 = (maxBound - origin) / dir;
        if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return java.util.Collections.emptyList();
    }
}

// 3. Now we have tMin (entry) and tMax (exit)
if (tMax < Ray.EPSILON) return java.util.Collections.emptyList();
if (tMin < Ray.EPSILON) tMin = tMax; // Ray inside the box

// 4. Create Intersection objects
Point3 pointIn = ray.pointAtParameter(tMin);
Vector3 normalIn = getNormalAt(pointIn);
Intersection in = new Intersection(pointIn, normalIn, tMin, this);

Point3 pointOut = ray.pointAtParameter(tMax);
Vector3 normalOut = getNormalAt(pointOut);
Intersection out = new Intersection(pointOut, normalOut, tMax, this);

// 5. Return the interval
return java.util.Arrays.asList(new IntersectionInterval(tMin, tMax, in,
out));
}

/**
 * Returns the surface normal at a given point on the prism's surface.
 * The point is in world coordinates. The normal is calculated in local
space
 * and then transformed back to world space.
 *
 * @param worldPoint The point on the prism's surface in world
coordinates.
 * @return The normalized normal vector at that point in world
coordinates.
 */

```

```

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // Ensure transforms are not null before proceeding
    if (inverseTransform == null || inverseTransposeTransformForNormal
        == null) {
        System.err.println("Error: Box transforms are null. Cannot compute
normal.");
        // Fallback: return a default normal or throw an exception
        return new Vector3(0, 1, 0);
    }

    // Transform the world point to the prism's local space
    Point3 localPoint =
this.getInverseTransform().transformPoint(worldPoint);

    Vector3 localNormal;

    // Use a slightly larger epsilon for normal calculation to avoid ambiguity
    at edges/corners
    double normalEpsilon = Ray.EPSILON * 10;

    double halfWidth = width / 2.0;
    double halfHeight = height / 2.0;
    double halfDepth = depth / 2.0;

    // Determine which face the local point is on by checking which
    coordinate is closest to the boundary
    if (Math.abs(localPoint.x - halfWidth) < normalEpsilon) {
        localNormal = new Vector3(1, 0, 0);
    } else if (Math.abs(localPoint.x + halfWidth) < normalEpsilon) {
        localNormal = new Vector3(-1, 0, 0);
    }
    else if (Math.abs(localPoint.y - halfHeight) < normalEpsilon) {
        localNormal = new Vector3(0, 1, 0);
    } else if (Math.abs(localPoint.y + halfHeight) < normalEpsilon) {
        localNormal = new Vector3(0, -1, 0);
    }
    else if (Math.abs(localPoint.z - halfDepth) < normalEpsilon) {
        localNormal = new Vector3(0, 0, 1);
    }
}

```

```

} else if (Math.abs(localPoint.z + halfDepth) < normalEpsilon) {
    localNormal = new Vector3(0, 0, -1);
} else {
    // Fallback for floating point inaccuracies near edges/corners.
    // This attempts to find the closest face based on the local point's
    coordinates.
    double[] dists = {
        Math.abs(localPoint.x - halfWidth),
        Math.abs(localPoint.x + halfWidth),
        Math.abs(localPoint.y - halfHeight),
        Math.abs(localPoint.y + halfHeight),
        Math.abs(localPoint.z - halfDepth),
        Math.abs(localPoint.z + halfDepth)
    };
    int minIdx = 0;
    for(int i = 1; i < 6; i++) {
        if (dists[i] < dists[minIdx]) {
            minIdx = i;
        }
    }
    switch(minIdx) {
        case 0: localNormal = new Vector3(1, 0, 0); break;
        case 1: localNormal = new Vector3(-1, 0, 0); break;
        case 2: localNormal = new Vector3(0, 1, 0); break;
        case 3: localNormal = new Vector3(0, -1, 0); break;
        case 4: localNormal = new Vector3(0, 0, 1); break;
        case 5: localNormal = new Vector3(0, 0, -1); break;
        default: localNormal = new Vector3(0, 1, 0); // Should not happen
    }
    System.err.println("Warning: Box normal fallback used due to floating
    point inaccuracy.");
}

// Transform the local normal back to world space
// Use inverse transpose for correct normal transformation
return
this.inverseTransposeTransformForNormal.transformVector(localNormal)
.normalize();
}

```

```
}
```

```
// =====
// File: /net/elenamurat/shape/Sphere.java
// =====

package net.elena.murat.shape;

import java.awt.Color;
import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;
import net.elena.murat.material.SolidColorMaterial;

public class Sphere implements EMShape {
    private Material material;

    // Sphere's definition in its LOCAL coordinate system.
    // Canonical sphere: center at (0,0,0) and radius 'localRadius'.
    private final Point3 localCenter;
    private final double localRadius;

    // Transformation matrices
    private Matrix4 transform;      // Local to World transformation matrix
    private Matrix4 inverseTransform; // World to Local transformation
    matrix

    public Sphere(double radius) {
        this(radius, new SolidColorMaterial(Color.BLUE));
    }

    public Sphere(double radius, Material material) {
        // Define the sphere in its canonical local space: centered at origin
        this.localCenter = new Point3(0, 0, 0);
        this.localRadius = radius;
    }
}
```

```

this.material = material;

// Initialize with identity transform by default
this.transform = new Matrix4();
this.inverseTransform = new Matrix4();
}

// The getCenter() method is no longer directly applicable for the world
position
// if using transformations. You'd calculate it from the transform matrix.
// However, it can still return the local center for internal use.
public Point3 getLocalCenter() {
    return localCenter;
}

// --- EMShape Interface Implementations ---

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

/**
 * Sets the transformation matrix that converts points/vectors from the
sphere's
 * local space to world space. When this is set, the inverse transform is
also computed.
 * @param transform The 4x4 transformation matrix.
 */
@Override
public void setTransform(Matrix4 transform) {
    this.transform = transform;
    this.inverseTransform = transform.inverse(); // Pre-compute inverse for
efficiency
}

```

```
}

/***
 * Returns the transformation matrix that converts points/vectors from the
sphere's
 * local space to world space.
 * @return The 4x4 transformation matrix.
*/
@Override
public Matrix4 getTransform() {
    return this.transform;
}

/***
 * Returns the inverse of the transformation matrix, which converts
points/vectors
 * from world space back to the sphere's local space.
 * @return The 4x4 inverse transformation matrix.
*/
@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}

/***
 * Finds the intersection of a ray with this Sphere.
 * The ray is first transformed into the sphere's local coordinate system
for intersection testing.
 * @param ray The ray to test intersection, in world coordinates.
 * @return The t value where the ray intersects the sphere, or
Double.POSITIVE_INFINITY if no intersection.
*/
@Override
public double intersect(Ray ray) {
    // 1. Transform the ray into the sphere's local coordinate system
    // This effectively transforms the problem from intersecting a
transformed sphere
    // with a world-space ray, to intersecting a canonical sphere with a
    // locally-transformed ray.
```

```

Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
Vector3 localDirection =
inverseTransform.transformVector(ray.getDirection()).normalize(); //
Normalize after transform

// Use local references for calculations to simplify
Vector3 oc = localOrigin.subtract(localCenter); // localCenter is (0,0,0),
so this is just localOrigin

double a = localDirection.dot(localDirection);
double b = 2.0 * oc.dot(localDirection);
double c = oc.dot(oc) - localRadius * localRadius;
double discriminant = b * b - 4 * a * c;

if (discriminant < 0) {
    return Double.POSITIVE_INFINITY;
} else {
    double sqrtDiscriminant = Math.sqrt(discriminant);

    // First intersection point
    double t1 = (-b - sqrtDiscriminant) / (2.0 * a);
    // Second intersection point
    double t2 = (-b + sqrtDiscriminant) / (2.0 * a);

    // Find the closest valid intersection (t > Ray.EPSILON)
    if (t1 > Ray.EPSILON && t2 > Ray.EPSILON) {
        return Math.min(t1, t2);
    } else if (t1 > Ray.EPSILON) {
        return t1;
    } else if (t2 > Ray.EPSILON) {
        return t2;
    }
    return Double.POSITIVE_INFINITY; // No valid intersection in front
of the ray
}

/***
* Calculates all intersection intervals between a ray and this sphere.

```

```

* The ray is transformed into the sphere's local space for calculation.
* For a sphere, there are typically two intersection points (in and out),
* forming a single interval where the ray is inside the sphere.
* @param ray The ray to test, in world coordinates.
* @return A list containing the intersection interval(s). Empty if no
intersection.
*/
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into the sphere's local coordinate system
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
        inverseTransform.transformVector(ray.getDirection()).normalize();

    // Use local references
    Vector3 oc = localOrigin.subtract(localCenter); // localCenter is (0,0,0)

    double a = localDirection.dot(localDirection);
    double b = 2.0 * oc.dot(localDirection);
    double c = oc.dot(oc) - localRadius * localRadius;
    double discriminant = b * b - 4 * a * c;

    if (discriminant < 0) {
        return java.util.Collections.emptyList();
    }

    double sqrtDiscriminant = Math.sqrt(discriminant);
    double t1 = (-b - sqrtDiscriminant) / (2.0 * a);
    double t2 = (-b + sqrtDiscriminant) / (2.0 * a);

    // Ensure t1 is the entry, t2 is the exit
    if (t1 > t2) {
        double temp = t1;
        t1 = t2;
        t2 = temp;
    }

    // Only consider intersections in front of the ray
    boolean t1Valid = t1 > Ray.EPSILON;
}

```

```

boolean t2Valid = t2 > Ray.EPSILON;

if (!t1Valid && !t2Valid) {
    return java.util.Collections.emptyList();
}

// Create Intersection objects
Point3 point1 = ray.pointAtParameter(t1);
Vector3 normal1 = getNormalAt(point1);
Intersection in = new Intersection(point1, normal1, t1, this);

Point3 point2 = ray.pointAtParameter(t2);
Vector3 normal2 = getNormalAt(point2);
Intersection out = new Intersection(point2, normal2, t2, this);

// Return a single interval
return java.util.Arrays.asList(new IntersectionInterval(t1, t2, in, out));
}

/**
 * Calculates the normal vector at a given point on the Sphere's surface in
 * WORLD coordinates.
 * This involves transforming the world hit point to local space,
 * calculating the local normal,
 * and then transforming it back to world space using the inverse
 * transpose of the model matrix.
 * @param worldPoint The point on the Sphere's surface in world
 * coordinates.
 * @return The normalized normal vector at that point in world
 * coordinates.
 */
@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // 1. Transform the world hit point to the sphere's local coordinate
    // system
    Point3 localHitPoint = inverseTransform.transformPoint(worldPoint);

    // 2. Calculate the normal in local space. For a sphere centered at
    localCenter (0,0,0),
}

```

```

    // the normal is simply the normalized vector from the localCenter to the
    localHitPoint.

    Vector3 localNormal = localHitPoint.subtract(localCenter).normalize();

    // 3. Transform the local normal back to world space.
    // Normals transform with the inverse transpose of the model matrix.
    //Matrix4 normalTransformMatrix =
this.inverseTransform.transpose(); // M_normal = (M^-1)^T
    Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal(); // Normaller için
yeni metod
    return
normalTransformMatrix.transformVector(localNormal).normalize(); //
Ensure normalized after transform
}
}

```

```

// =====
// File: /net/elenamurat/shape/Ellipsoid.java
// =====

```

```

package net.elena.murat.shape;

import java.util.List;
import java.util.Objects;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

public class Ellipsoid implements EMShape {
    private final Point3 center;
    private final double a, b, c;
    private Matrix4 transform;
    private Matrix4 inverseTransform;
    private Matrix4 inverseTransposeTransform;
    private Material material;

    public Ellipsoid(Point3 center, double a, double b, double c) {

```

```

this.center = Objects.requireNonNull(center);
this.a = a;
this.b = b;
this.c = c;
this.transform = Matrix4.identity();
this.inverseTransform = Matrix4.identity();
this.inverseTransposeTransform = Matrix4.identity();
}

@Override
public double intersect(Ray ray) {
    Ray localRay = transformRayToLocalSpace(ray);

    double a2 = a * a;
    double b2 = b * b;
    double c2 = c * c;

    Vector3 o = localRay.getOrigin().toVector3();
    Vector3 d = localRay.getDirection();

    double A = (d.x*d.x)/a2 + (d.y*d.y)/b2 + (d.z*d.z)/c2;
    double B = 2*((o.x*d.x)/a2 + (o.y*d.y)/b2 + (o.z*d.z)/c2);
    double C = (o.x*o.x)/a2 + (o.y*o.y)/b2 + (o.z*o.z)/c2 - 1;

    double discriminant = B*B - 4*A*C;
    if (discriminant < 0) return -1;

    double sqrtD = Math.sqrt(discriminant);
    double t1 = (-B - sqrtD)/(2*A);
    double t2 = (-B + sqrtD)/(2*A);

    return (t1 > Ray.EPSILON) ? t1 : (t2 > Ray.EPSILON) ? t2 : -1;
}

/**
 * Calculates all intersection intervals between a ray and this ellipsoid.
 * The ray is transformed into the ellipsoid's local space for calculation.
 * The method solves the quadratic equation for the ellipsoid and returns
 * an interval where the ray is inside the shape.

```

```

* @param ray The ray to test, in world coordinates.
* @return A list of IntersectionInterval objects. Empty if no intersection.
*/
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into local space
    Ray localRay = transformRayToLocalSpace(ray);

    double a2 = a * a;
    double b2 = b * b;
    double c2 = c * c;

    Vector3 o = localRay.getOrigin().toVector3();
    Vector3 d = localRay.getDirection();

    double A = (d.x*d.x)/a2 + (d.y*d.y)/b2 + (d.z*d.z)/c2;
    double B = 2*((o.x*d.x)/a2 + (o.y*d.y)/b2 + (o.z*d.z)/c2);
    double C = (o.x*o.x)/a2 + (o.y*o.y)/b2 + (o.z*o.z)/c2 - 1;

    double discriminant = B*B - 4*A*C;
    if (discriminant < 0) {
        return java.util.Collections.emptyList();
    }

    double sqrtD = Math.sqrt(discriminant);
    double t1 = (-B - sqrtD)/(2*A);
    double t2 = (-B + sqrtD)/(2*A);

    // Ensure t1 is entry, t2 is exit
    if (t1 > t2) {
        double temp = t1;
        t1 = t2;
        t2 = temp;
    }

    // Only consider intersections in front of the ray
    boolean t1Valid = t1 > Ray.EPSILON;
    boolean t2Valid = t2 > Ray.EPSILON;
}

```

```

if (!t1Valid && !t2Valid) {
    return java.util.Collections.emptyList();
}

// Create Intersection objects
Point3 pointIn = ray.pointAtParameter(t1);
Vector3 normalIn = getNormalAt(pointIn);
Intersection in = new Intersection(pointIn, normalIn, t1, this);

Point3 pointOut = ray.pointAtParameter(t2);
Vector3 normalOut = getNormalAt(pointOut);
Intersection out = new Intersection(pointOut, normalOut, t2, this);

// Return a single interval
return java.util.Arrays.asList(new IntersectionInterval(t1, t2, in, out));
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    Point3 localPoint = inverseTransform.transformPoint(worldPoint);
    Vector3 localNormal = new Vector3(
        localPoint.x/(a*a),
        localPoint.y/(b*b),
        localPoint.z/(c*c)
    ).normalize();

    return
    inverseTransposeTransform.transformVector(localNormal).normalize();
}

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setMaterial(Material material) {
    this.material = Objects.requireNonNull(material);
}

```

```
@Override
public void setTransform(Matrix4 transform) {
    this.transform = Objects.requireNonNull(transform);
    this.inverseTransform = transform.inverse();
    this.inverseTransposeTransform = inverseTransform.transpose();
}

@Override
public Matrix4 getTransform() {
    return transform;
}

@Override
public Matrix4 getInverseTransform() {
    return inverseTransform;
}

private Ray transformRayToLocalSpace(Ray worldRay) {
    Point3 localOrigin =
    inverseTransform.transformPoint(worldRay.getOrigin());
    Vector3 localDirection =
    inverseTransform.transformVector(worldRay.getDirection()).normalize();
    return new Ray(localOrigin, localDirection);
}

// =====
// File: /net/elenamurat/shape/Triangle.java
// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.material.Material;
import net.elena.murat.math.*;
```

```

/***
 * Triangle class represents a single triangle, defined by three vertices.
 * It implements the EMShape interface, supporting Matrix4
transformations.
 * The vertices are defined in the triangle's local coordinate system.
*/
public class Triangle implements EMShape {
    // Vertices defined in the triangle's LOCAL coordinate system
    private final Point3 localV0, localV1, localV2;
    private Material material;

    // Transformation matrices
    private Matrix4 transform;      // Local to World transformation matrix
    private Matrix4 inverseTransform; // World to Local transformation
matrix

    // Precomputed local normal for optimization (recalculated if vertices
change, not transform)
    private Vector3 precomputedLocalNormal;

    /**
     * Constructs a triangle using its three vertices in its LOCAL coordinate
system.
     * The material must be set separately using setMaterial().
     * The transformation matrix is initialized to identity; use setTransform()
to position.
     * @param v0 The first vertex in local space.
     * @param v1 The second vertex in local space.
     * @param v2 The third vertex in local space.
    */
    public Triangle(Point3 v0, Point3 v1, Point3 v2) {
        this.localV0 = v0;
        this.localV1 = v1;
        this.localV2 = v2;
        this.material = null;

        // Initialize with identity transform by default
        this.transform = new Matrix4();
        this.inverseTransform = new Matrix4();
    }
}

```

```
    precomputeLocalNormal(); // Compute normal based on local vertices
}
```

```
// --- EMShape Interface Implementations ---
```

```
@Override
public void setMaterial(Material material) {
    this.material = material;
}
```

```
@Override
public Material getMaterial() {
    return this.material;
}
```

```
/**
 * Sets the transformation matrix that converts points/vectors from the
triangle's
 * local space to world space. When this is set, the inverse transform is
also computed.
```

```
 * @param transform The 4x4 transformation matrix.
*/
```

```
@Override
public void setTransform(Matrix4 transform) {
    this.transform = transform;
    this.inverseTransform = transform.inverse(); // Pre-compute inverse for
efficiency
```

```
    // You might want to add a check here if inverseTransform is null
(determinant zero),
```

```
    // similar to your Matrix3 constructor, though Matrix4.inverse() should
handle it.
```

```
    if (this.inverseTransform == null) {
        System.err.println("Warning: Could not compute inverse transform for
Triangle (determinant zero). Using identity matrix.");
        this.inverseTransform = new Matrix4();
    }
}
```

```

/***
 * Returns the transformation matrix that converts points/vectors from the
triangle's
 * local space to world space.
 * @return The 4x4 transformation matrix.
*/
@Override
public Matrix4 getTransform() {
    return this.transform;
}

/***
 * Returns the inverse of the transformation matrix, which converts
points/vectors
 * from world space back to the triangle's local space.
 * @return The 4x4 inverse transformation matrix.
*/
@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}

/***
 * Precomputes the normal vector of the triangle in its LOCAL coordinate
system.
 * This normal assumes a counter-clockwise winding order of vertices
(v0, v1, v2)
 * when viewed from the front face.
*/
private void precomputeLocalNormal() {
    Vector3 edge1 = localV1.subtract(localV0);
    Vector3 edge2 = localV2.subtract(localV0);
    this.precomputedLocalNormal = edge1.cross(edge2).normalize();
}

/***
 * Calculates the intersection of a ray with this triangle.
 * The ray is first transformed into the triangle's local coordinate system

```

for intersection testing.

* Uses the Moller-Trumbore algorithm for efficient ray-triangle intersection.

* @param ray The ray to test intersection, in world coordinates.

* @return The t value where the ray intersects the triangle, or Double.POSITIVE_INFINITY if no intersection.

*/

@Override

public double intersect(Ray ray) {

// 1. Transform the world ray into the triangle's local coordinate system

Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());

// Normalize localDirection in case of non-uniform scaling affecting length,

// although for Moller-Trumbore, the length of the direction vector affects 't' linearly.

// It's crucial for normal transformations later if using transformVector with non-uniform scale.

Vector3 localDirection =

inverseTransform.transformVector(ray.getDirection()).normalize();

// Create a local ray for intersection testing

Ray localRay = new Ray(localOrigin, localDirection);

// Vertices for intersection are already in local space (localV0, localV1, localV2)

Vector3 edge1 = localV1.subtract(localV0);

Vector3 edge2 = localV2.subtract(localV0);

Vector3 pvec = localRay.getDirection().cross(edge2);

double det = edge1.dot(pvec);

// Check for parallel ray (determinant close to zero)

if (det > -Ray.EPSILON && det < Ray.EPSILON) {

 return Double.POSITIVE_INFINITY;

}

double invDet = 1.0 / det;

Vector3 tvec = localRay.getOrigin().subtract(localV0);

```

double u = tvec.dot(pvec) * invDet;

// Check barycentric coordinate U
if (u < -Ray.EPSILON || u > 1.0 + Ray.EPSILON) { // Add epsilon for
robustness
    return Double.POSITIVE_INFINITY;
}

Vector3 qvec = tvec.cross(edge1);
double v = localRay.getDirection().dot(qvec) * invDet;

// Check barycentric coordinate V
if (v < -Ray.EPSILON || u + v > 1.0 + Ray.EPSILON) { // Add epsilon
for robustness
    return Double.POSITIVE_INFINITY;
}

double t = edge2.dot(qvec) * invDet;

// Check if the intersection point is in front of the ray origin
if (t > Ray.EPSILON) {
    return t;
} else {
    return Double.POSITIVE_INFINITY;
}
}

/***
 * Calculates all intersection intervals between a ray and this triangle.
 * Since a triangle is a flat, infinitely thin surface, the entry and exit
points are considered the same.
 * @param ray The ray to test, in world coordinates.
 * @return A list containing a single degenerate interval if intersected,
otherwise empty list.
 */
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into the triangle's local coordinate system
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());

```

```

Vector3 localDirection =
inverseTransform.transformVector(ray.getDirection()).normalize();
Ray localRay = new Ray(localOrigin, localDirection);

// 2. Use Moller-Trumbore algorithm to find intersection
Vector3 edge1 = localV1.subtract(localV0);
Vector3 edge2 = localV2.subtract(localV0);
Vector3 pvec = localDirection.cross(edge2);
double det = edge1.dot(pvec);

// Check for parallel ray
if (Math.abs(det) < Ray.EPSILON) {
    return java.util.Collections.emptyList();
}

double invDet = 1.0 / det;
Vector3 tvec = localOrigin.subtract(localV0);
double u = tvec.dot(pvec) * invDet;

// Barycentric coordinate u check
if (u < -Ray.EPSILON || u > 1.0 + Ray.EPSILON) {
    return java.util.Collections.emptyList();
}

Vector3 qvec = tvec.cross(edge1);
double v = localDirection.dot(qvec) * invDet;

// Barycentric coordinate v check
if (v < -Ray.EPSILON || u + v > 1.0 + Ray.EPSILON) {
    return java.util.Collections.emptyList();
}

double t = edge2.dot(qvec) * invDet;

// Check if intersection is in front of the ray
if (t <= Ray.EPSILON) {
    return java.util.Collections.emptyList();
}

```

```

// 3. Create a single degenerate interval
Point3 worldHit = ray.pointAtParameter(t);
Vector3 worldNormal = getNormalAt(worldHit);
Intersection hit = new Intersection(worldHit, worldNormal, t, this);

IntersectionInterval interval = IntersectionInterval.point(t, hit);
return java.util.Arrays.asList(interval);
}

/**
 * Calculates the normal vector at a given point on the Triangle's surface
in WORLD coordinates.
 * For a triangle, the normal is constant across its surface (assuming it's
flat).
 * This involves transforming the precomputed local normal to world
space using the
 * inverse transpose of the model matrix.
 * @param worldPoint The point on the Triangle's surface in world
coordinates (can be ignored for flat triangles).
 * @return The normalized normal vector at that point in world
coordinates.
*/
@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // Normals transform with the inverse transpose of the model matrix.
    // For triangles, the normal is constant across the surface.
    //Matrix4 normalTransformMatrix =
this.inverseTransform.transpose(); // M_normal = (M^-1)^T
    Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal(); // Normaller için
yeni metod
    return
normalTransformMatrix.transformVector(precomputedLocalNormal).nor
malize(); // Ensure normalized after transform
}

// You might still want these getters for debugging or specific use cases,
// but remember they return vertices in LOCAL space.
public Point3 getLocalV0() {

```

```
    return localV0;
}

public Point3 getLocalV1() {
    return localV1;
}

public Point3 getLocalV2() {
    return localV2;
}
}
```

```
// =====
// File: /net/elenamurat/shape/EmojiBillboard.java
// =====
```

```
package net.elena.murat.shape;

import java.awt.image.BufferedImage;
import java.util.List;

import net.elena.murat.material.Material;
import net.elena.murat.math.*;

/**
 * A 2D quad in 3D space for displaying transparent images (e.g., emojis).
 * No UV passed to Intersection. Material must compute UV from point
and transform.
 */
public class EmojiBillboard implements EMShape {

    private final double width;
    private final double height;
    private final boolean isRectangle;
    private final boolean isVisible;
    private final BufferedImage texture;

    private Matrix4 transform = Matrix4.identity();
```

```
private Matrix4 inverseTransform;
private Matrix4 inverseTransposeTransformForNormal;
private Material material;

public EmojiBillboard(double width, double height,
    boolean isRectangle,
    boolean isVisible,
    BufferedImage texture) {
    this.width = width;
    this.height = height;
    this.isRectangle = isRectangle;
    this.isVisible = isVisible;
    this.texture = texture;

    updateTransforms();
}

public EmojiBillboard(double width, double height) {
    this(width, height, true, true, null);
}

public EmojiBillboard(double size) {
    this(size, size);
}

public EmojiBillboard() {
    this(1.0, 1.0);
}

// --- EMShape Methods ---

@Override
public void setTransform(Matrix4 transform) {
    this.transform = new Matrix4(transform);
    updateTransforms();
}

@Override
public Matrix4 getTransform() {
```

```
    return this.transform;
}

@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}

private void updateTransforms() {
    this.inverseTransform = this.transform.inverse();
    this.inverseTransposeTransformForNormal =
this.transform.inverse().transpose();
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

@Override
public Material getMaterial() {
    return this.material;
}

public boolean isVisible() {
    return isVisible;
}

@Override
public double intersect(Ray ray) {
    if (isRectangle) {
        return intersectR(ray);
    } else {
        return intersectO(ray);
    }
}

// --- Original intersect ---
public double intersectR(Ray ray) {
```

```

if (inverseTransform == null) return Double.POSITIVE_INFINITY;

Ray localRay = ray.transform(inverseTransform);

Vector3 dir = localRay.getDirection();
if (Math.abs(dir.z) < Ray.EPSILON) {
    return Double.POSITIVE_INFINITY;
}

double t = -localRay.getOrigin().z / dir.z;
if (t < Ray.EPSILON) return Double.POSITIVE_INFINITY;

Point3 localHit = localRay.pointAtParameter(t);
double x = localHit.x;
double y = localHit.y;

double halfWidth = width / 2.0;
double halfHeight = height / 2.0;

    if (Math.abs(x) <= halfWidth && Math.abs(y) <= halfHeight) {
        return t;
    }

return Double.POSITIVE_INFINITY;
}

public double intersectO(Ray ray) {
    if (inverseTransform == null) return Double.POSITIVE_INFINITY;

    Ray localRay = ray.transform(inverseTransform);

    Vector3 dir = localRay.getDirection();
    if (Math.abs(dir.z) < Ray.EPSILON) {
        return Double.POSITIVE_INFINITY;
    }

    double t = -localRay.getOrigin().z / dir.z;
    if (t < Ray.EPSILON) return Double.POSITIVE_INFINITY;
}

```

```

Point3 localHit = localRay.pointAtParameter(t);
double x = localHit.x;
double y = localHit.y;

double radiusX = width / 2.0;
double radiusY = height / 2.0;

// Normalize coordinates to unit circle
double nx = x / radiusX;
double ny = y / radiusY;

// Check if inside ellipse
if (nx * nx + ny * ny <= 1.0) {
    return t;
}

return Double.POSITIVE_INFINITY;
}

@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    double t = intersect(ray);
    if (t == Double.POSITIVE_INFINITY) {
        return java.util.Collections.emptyList();
    }

    Point3 hitPoint = ray.pointAtParameter(t);
    Vector3 normal = getNormalAt(hitPoint);

    Intersection intersection = new Intersection(hitPoint, normal, t, this);

    return java.util.Arrays.asList(
        new IntersectionInterval(t, t, intersection, intersection)
    );
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    if (inverseTransposeTransformForNormal == null) {

```

```

        return new Vector3(0, 0, 1);
    }

    Vector3 localNormal = new Vector3(0, 0, 1);
    return inverseTransposeTransformForNormal
        .transformDirection(localNormal)
        .normalize();
}

public double getWidth() {
    return width;
}

public double getHeight() {
    return height;
}

}

```

```

// =====
// File: /net/elenamurat/shape/CSGShape.java
// =====

```

```

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

public abstract class CSGShape implements EMShape {
    protected final EMShape left;
    protected final EMShape right;

    private Matrix4 transform;
    private Matrix4 inverseTransform;

    public CSGShape(EMShape left, EMShape right) {

```

```
this.left = left;
this.right = right;
this.transform = Matrix4.identity();
this.inverseTransform = Matrix4.identity();
}
```

```
@Override
public void setTransform(Matrix4 transform) {
    this.transform = new Matrix4(transform);
    this.inverseTransform = transform.inverse();
}
```

```
@Override
public Matrix4 getTransform() {
    return this.transform;
}
```

```
@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}
```

```
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Ray to CSG's local space
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
    inverseTransform.transformVector(ray.getDirection()).normalize();
    Ray localRay = new Ray(localOrigin, localDirection);

    // 2. Intersects of inner shapes
    List<IntersectionInterval> a = left.intersectAll(localRay);
    List<IntersectionInterval> b = right.intersectAll(localRay);

    // 3. Combine
    return combine(a, b);
}
```

```
protected abstract List<IntersectionInterval> combine(
```

```

List<IntersectionInterval> a,
List<IntersectionInterval> b
);

@Override
public double intersect(Ray ray) {
    List<IntersectionInterval> intervals = intersectAll(ray);
    return intervals.isEmpty() ? -1 : intervals.get(0).tIn;
}

/**
 * Calculates the normal vector at a given point on the CSG shape's
surface in WORLD coordinates.
 * This is a placeholder implementation that uses the left operand's
normal.
 * For accurate results, CSG operations should calculate normals based
on the surface hit.
 * @param worldPoint The point on the CSG shape's surface in world
coordinates.
 * @return The normalized normal vector at that point in world
coordinates.
*/
@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // 1. Transform the world point to the CSG operation's local space
    Point3 localPoint = inverseTransform.transformPoint(worldPoint);

    // 2. Determine which operand was hit and use its normal
    // This is a simple heuristic: check the distance to the surfaces of left and
right
    // A more accurate method would be to know which interval was hit in
intersectAll
    Vector3 normalA = left.getNormalAt(localPoint);
    Vector3 normalB = right.getNormalAt(localPoint);

    // Calculate the distance from the localPoint to the surfaces of A and B
    // This is a crude approximation using the dot product with the normal
    // A point is "on" a surface if it's very close to it
    double distA =

```

```

Math.abs(localPoint.subtract(left.getTransform().transformPoint(new
Point3(0,0,0))).dot(normalA));
    double distB =
Math.abs(localPoint.subtract(right.getTransform().transformPoint(new
Point3(0,0,0))).dot(normalB));

Vector3 localNormal;
if (distA < distB) {
    // Hit on A's surface
    localNormal = normalA;
} else {
    // Hit on B's surface
    // For DifferenceCSG, if we're on B's surface, the normal should point
inwards
    // because B is being subtracted
    if (this instanceof DifferenceCSG) {
        localNormal = normalB.negate();
    } else {
        localNormal = normalB;
    }
}

// 3. Transform the local normal back to world space
Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal();
return
normalTransformMatrix.transformVector(localNormal).normalize();
}

@Override
public Material getMaterial() {
    return left.getMaterial();
}

@Override
public void setMaterial(Material material) {
    left.setMaterial(material);
    right.setMaterial(material);
}

```

```
}
```

```
// =====  
// File: /net/elenamurat/shape/IntersectionCSG.java  
// =====
```

```
package net.elena.murat.shape;
```

```
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;
```

```
import net.elena.murat.math.*;
```

```
/**
```

```
 * Represents a Constructive Solid Geometry (CSG) Intersection  
operation.
```

```
 * The resulting shape is the intersection of two shapes: A ∩ B.
```

```
 * A point is inside the intersection only if it is inside both shape A and  
shape B.
```

```
 */
```

```
public class IntersectionCSG extends CSGShape {
```

```
/**
```

```
 * Constructs an IntersectionCSG from two shapes.
```

```
 * @param left The first shape (left operand).
```

```
 * @param right The second shape (right operand).
```

```
 */
```

```
public IntersectionCSG(EMShape left, EMShape right) {
```

```
    super(left, right);
```

```
}
```

```
/**
```

```
 * Combines the intersection intervals of two shapes using the  
Intersection operation.
```

```
 * The intersection is formed by finding intervals where the ray is inside  
both shapes.
```

```

* @param a List of intervals from the left shape.
* @param b List of intervals from the right shape.
* @return The resulting list of intervals for the intersection.
*/
@Override
protected List<IntersectionInterval> combine(
    List<IntersectionInterval> a,
    List<IntersectionInterval> b) {

    // 1. If either list is empty, intersection is empty
    if (a.isEmpty() || b.isEmpty()) {
        return Collections.emptyList();
    }

    // 2. Sort both interval lists by tIn
    List<IntersectionInterval> sortedA = new ArrayList<>(a);
    List<IntersectionInterval> sortedB = new ArrayList<>(b);
    Collections.sort(sortedA, (ia, ib) -> Double.compare(ia.tIn, ib.tIn));
    Collections.sort(sortedB, (ia, ib) -> Double.compare(ia.tIn, ib.tIn));

    List<IntersectionInterval> result = new ArrayList<>();

    int i = 0, j = 0;
    while (i < sortedA.size() && j < sortedB.size()) {
        IntersectionInterval intervalA = sortedA.get(i);
        IntersectionInterval intervalB = sortedB.get(j);

        // Find overlap: max(tIn) to min(tOut)
        double overlapTIn = Math.max(intervalA.tIn, intervalB.tIn);
        double overlapTOut = Math.min(intervalA.tOut, intervalB.tOut);

        // If there is a valid overlap
        if (overlapTIn < overlapTOut - Ray.EPSILON) {
            // Calculate midpoint of the two entry points
            Point3 pointIn = new Point3(
                (intervalA.in.getPoint().x + intervalB.in.getPoint().x) * 0.5,
                (intervalA.in.getPoint().y + intervalB.in.getPoint().y) * 0.5,
                (intervalA.in.getPoint().z + intervalB.in.getPoint().z) * 0.5
            );
        }
    }
}

```

```

Point3 pointOut = new Point3(
    (intervalA.out.getPoint().x + intervalB.out.getPoint().x) * 0.5,
    (intervalA.out.getPoint().y + intervalB.out.getPoint().y) * 0.5,
    (intervalA.out.getPoint().z + intervalB.out.getPoint().z) * 0.5
);

// Average the normals
Vector3 normalIn = intervalA.in.getNormal()
    .add(intervalB.in.getNormal())
    .normalize();
Vector3 normalOut = intervalA.out.getNormal()
    .add(intervalB.out.getNormal())
    .normalize();

Intersection in = new Intersection(pointIn, normalIn, overlapTIn,
this);
Intersection out = new Intersection(pointOut, normalOut,
overlapTOut, this);

result.add(new IntersectionInterval(overlapTIn, overlapTOut, in,
out));
}

// Advance the interval with the smaller tOut
if (intervalA.tOut < intervalB.tOut) {
    i++;
} else {
    j++;
}
}

return result;
}

}

// =====
// File: /net/elenamurat/shape/Rectangle3D.java

```

```

// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

import static net.elena.murat.math.Vector3.*;

public class Rectangle3D implements EMShape {
    private final Point3 p1, p2;
    private final float thickness;
    private Material material;
    private Matrix4 transform = Matrix4.identity();
    private Matrix4 inverseTransform = Matrix4.identity();

    public Rectangle3D(Point3 p1, Point3 p2, float thickness) {
        this.p1 = p1;
        this.p2 = p2;
        this.thickness = thickness;
    }

    @Override
    public double intersect(Ray ray) {
        // Ray to local space
        Ray localRay = new Ray(
            inverseTransform.transformPoint(ray.getOrigin()),
            inverseTransform.transformVector(ray.getDirection())
        );

        // plane: normal = (0,0,1)
        if (Math.abs(localRay.getDirection().z) < Ray.EPSILON) return -1;

        double t = -localRay.getOrigin().z / localRay.getDirection().z;
        if (t < Ray.EPSILON) return -1;

        Point3 localHit = localRay.pointAtParameter(t);

```

```

    if (localHit.x < Math.min(p1.x, p2.x) || localHit.x > Math.max(p1.x,
p2.x) ||
        localHit.y < Math.min(p1.y, p2.y) || localHit.y > Math.max(p1.y,
p2.y)) {
            return -1;
        }

        return t;
    }

/***
 * Calculates all intersection intervals between a ray and this
Rectangle3D.
 * Since the rectangle is treated as a thin surface, the entry and exit points
are the same.
 * @param ray The ray to test, in world coordinates.
 * @return A list containing a single degenerate interval if intersected,
otherwise empty list.
 */
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize()
    );

    // 2. Check for intersection with the plane (z=0 in local space)
    if (Math.abs(localRay.getDirection().z) < Ray.EPSILON) {
        return java.util.Collections.emptyList(); // Parallel to the plane
    }

    double t = -localRay.getOrigin().z / localRay.getDirection().z;
    if (t <= Ray.EPSILON) {
        return java.util.Collections.emptyList(); // Behind the ray
    }

    // 3. Check if the hit point is within the rectangle bounds
    Point3 localHit = localRay.pointAtParameter(t);
}

```

```

double minX = Math.min(p1.x, p2.x);
double maxX = Math.max(p1.x, p2.x);
double minY = Math.min(p1.y, p2.y);
double maxY = Math.max(p1.y, p2.y);

if (localHit.x < minX || localHit.x > maxX || localHit.y < minY ||
localHit.y > maxY) {
    return java.util.Collections.emptyList();
}

// 4. Create a single degenerate interval
Point3 worldHit = ray.pointAtParameter(t);
Vector3 worldNormal = getNormalAt(worldHit);
Intersection hit = new Intersection(worldHit, worldNormal, t, this);

IntersectionInterval interval = IntersectionInterval.point(t, hit);
return java.util.Arrays.asList(interval);
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    Point3 localPoint = inverseTransform.transformPoint(worldPoint);
    Vector3 localNormal = new Vector3(0, 0, 1);
    return
inverseTransform.transpose().transformVector(localNormal).normalize();
}

@Override
public void setTransform(Matrix4 transform) {
    this.transform = transform;
    this.inverseTransform = transform.inverse();
}

@Override public Matrix4 getTransform() { return transform; }
@Override public Matrix4 getInverseTransform() { return
inverseTransform; }
@Override public Material getMaterial() { return material; }
@Override public void setMaterial(Material material) { this.material =
material; }

```

```
}
```

```
// =====
// File: /net/elenamurat/shape/Crescent.java
// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/**
 * A 3D crescent shape (hilal) formed by the subtraction of one sphere
 * from another.
 * Implements the EMShape interface for ray intersection and normal
 * calculation.
 * This implementation uses Constructive Solid Geometry (CSG) for
 * intersection.
 */
public class Crescent implements EMShape {
    private final double radius;      // Main sphere radius
    private final double cutRadius;   // Smaller sphere radius that cuts into
    the main sphere
    private final double cutDistance; // Distance between sphere centers
    (from main sphere center to cut sphere center)
    private Material material;
    private Matrix4 transform = Matrix4.identity();
    private Matrix4 inverseTransform = Matrix4.identity();
    private Matrix4 inverseTransposeTransform = Matrix4.identity(); // For
    transforming normals

    // Sphere centers in local space (main sphere at origin, cut sphere along
    X-axis)
    private final Point3 mainSphereCenter = new Point3(0, 0, 0);
    private final Point3 cutSphereCenter;
```

```

/***
 * Creates a crescent shape (hilal) using two spheres.
 * The crescent is formed by subtracting the 'cutSphere' from the
'mainSphere'.
 * @param radius Main sphere radius.
 * @param cutRadius Smaller sphere radius that cuts into the main
sphere.
 * @param cutDistance Distance between the center of the main sphere
(at origin)
 * and the center of the cut sphere (along X-axis).
 */
public Crescent (double radius, double cutRadius, double cutDistance) {
    this.radius = radius;
    this.cutRadius = cutRadius;
    // Ensure cutDistance is valid for a visible crescent.
    // It must be greater than radius - cutRadius (otherwise cut sphere is
fully inside main sphere)
    // and less than radius + cutRadius (otherwise spheres don't intersect).
    // We clamp it to a reasonable range.
    this.cutDistance = Math.max(Ray.EPSILON, Math.min(cutDistance,
radius + cutRadius - Ray.EPSILON));

    // Initialize the cut sphere's center. Main sphere is at origin.
    this.cutSphereCenter = new Point3(this.cutDistance, 0, 0);
}

@Override
public double intersect(Ray ray) {
    // Transform ray to local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()))
);

// Find all intersections with both spheres
// We need both entry and exit points for CSG
    double[] tMain = intersectSphereAll(localRay, mainSphereCenter,
radius);
    double[] tCut = intersectSphereAll(localRay, cutSphereCenter,

```

```

cutRadius);

// No intersection with main sphere means no crescent
if (tMain[0] < Ray.EPSILON && tMain[1] < Ray.EPSILON) {
    return -1;
}

double finalT = -1;

// CSG Logic: A AND NOT B (Main Sphere AND NOT Cut Sphere)
// Iterate through the intersection points of the main sphere
for (int i = 0; i < 2; i++) {
    double t = tMain[i];
    if (t > Ray.EPSILON) { // Check for valid positive intersection
        Point3 hitPoint = localRay.pointAtParameter(t); // Use
        pointAtParameter
        // Check if this point is outside the cut sphere
        if (hitPoint.distance(cutSphereCenter) >= cutRadius - Ray.EPSILON)
    } // Use distance()
    // This is a valid entry point into the crescent
    finalT = t;
    break; // Found the closest valid intersection
}
}

// If no valid intersection found yet, check if ray starts inside the
crescent
// (i.e., inside main sphere but outside cut sphere)
if (finalT < Ray.EPSILON) {
    boolean rayOriginInMain =
localRay.getOrigin().distance(mainSphereCenter) < radius -
Ray.EPSILON;
    boolean rayOriginOutCut =
localRay.getOrigin().distance(cutSphereCenter) >= cutRadius -
Ray.EPSILON;

    if (rayOriginInMain && rayOriginOutCut) {
        // Ray starts inside the crescent. Find the exit point from the main

```

sphere.

```
    if (tMain[1] > Ray.EPSILON) {
        finalT = tMain[1];
    }
}

return finalT;
}
```

```
/**
```

* Calculates all intersection intervals between a ray and this crescent shape.

* The crescent is defined as the region inside the main sphere and outside the cut sphere.

* This method returns a list of intervals where the ray is inside the crescent.

* @param ray The ray to test, in world coordinates.

* @return A list of IntersectionInterval objects. Empty if no intersection.

```
*/
```

@Override

```
public List<IntersectionInterval> intersectAll(Ray ray) {
```

```
    // 1. Transform the ray into local space
```

```
    Ray localRay = new Ray(
```

```
        inverseTransform.transformPoint(ray.getOrigin()),
```

```
        inverseTransform.transformVector(ray.getDirection()).normalize()
```

```
    );
```

```
    // 2. Get all intersections with both spheres
```

```
    double[] tMain = intersectSphereAll(localRay, mainSphereCenter,
```

```
    radius);
```

```
    double[] tCut = intersectSphereAll(localRay, cutSphereCenter,
```

```
    cutRadius);
```

```
    // 3. Collect all valid t values with their "in/out" status
```

```
    List<TimedHit> hits = new java.util.ArrayList<>();
```

```
    // Add main sphere intersections (in: +1, out: -1)
```

```
    if (tMain[0] > Ray.EPSILON) hits.add(new TimedHit(tMain[0], 1,
```

```

true));
    if (tMain[1] > Ray.EPSILON) hits.add(new TimedHit(tMain[1], -1,
true));

    // Add cut sphere intersections (in: +1, out: -1)
    if (tCut[0] > Ray.EPSILON) hits.add(new TimedHit(tCut[0], 1, false));
    if (tCut[1] > Ray.EPSILON) hits.add(new TimedHit(tCut[1], -1, false));

    // 4. Sort by t
    java.util.Collections.sort(hits, (a, b) -> Double.compare(a.t, b.t));

    // 5. Track state: insideMain, insideCut
    boolean insideMain = false;
    boolean insideCut = false;
    boolean wasInsideCrescent = false;
    double currentIntervalStart = -1;

List<IntersectionInterval> intervals = new java.util.ArrayList<>();

for (TimedHit hit : hits) {
    // Update state
    if (hit.isMain) {
        insideMain = hit.delta > 0 ? true : false;
    } else {
        insideCut = hit.delta > 0 ? true : false;
    }

    boolean isInsideCrescent = insideMain && !insideCut;

    // Entering crescent
    if (!wasInsideCrescent && isInsideCrescent) {
        currentIntervalStart = hit.t;
    }

    // Exiting crescent
    if (wasInsideCrescent && !isInsideCrescent && currentIntervalStart
    >= 0) {
        // Create Intersection objects
        Point3 pointIn = ray.pointAtParameter(currentIntervalStart);
    }
}

```

```

        Point3 pointOut = ray.pointAtParameter(hit.t);
        Vector3 normalIn = getNormalAt(pointIn);
        Vector3 normalOut = getNormalAt(pointOut);
        Intersection in = new Intersection(pointIn, normalIn,
currentIntervalStart, this);
        Intersection out = new Intersection(pointOut, normalOut, hit.t, this);
        intervals.add(new IntersectionInterval(currentIntervalStart, hit.t, in,
out));
        currentIntervalStart = -1;
    }

    wasInsideCrescent = isInsideCrescent;
}

// Handle case where ray ends inside crescent (no exit)
// For rendering, we might ignore this, or treat it as going to infinity.
// In a bounded scene, this is rare. We skip for now.

return intervals;
}

// Helper class to track intersection events
private static class TimedHit {
    final double t;
    final int delta; // +1 for entry, -1 for exit
    final boolean isMain; // true if from main sphere

    TimedHit(double t, int delta, boolean isMain) {
        this.t = t;
        this.delta = delta;
        this.isMain = isMain;
    }
}

/***
 * Helper method to intersect a ray with a sphere and return both t1 and
t2.
 * @param ray The ray to test
 * @param center Sphere center

```

```

* @param radius Sphere radius
* @return An array containing t1 and t2. Returns {-1, -1} if no
intersection.
*/
private double[] intersectSphereAll(Ray ray, Point3 center, double
radius) {
    Vector3 oc = ray.getOrigin().subtract(center);
    double a = ray.getDirection().dot(ray.getDirection());
    double b = 2.0 * oc.dot(ray.getDirection());
    double c = oc.dot(oc) - radius * radius;
    double discriminant = b * b - 4 * a * c;

    if (discriminant < 0) {
        return new double[]{-1, -1}; // No real roots
    }

    double sqrtDiscriminant = Math.sqrt(discriminant);
    double t1 = (-b - sqrtDiscriminant) / (2.0 * a);
    double t2 = (-b + sqrtDiscriminant) / (2.0 * a);

    // Ensure t1 is always the smaller (closer) positive intersection
    if (t1 > t2) {
        double temp = t1;
        t1 = t2;
        t2 = temp;
    }

    // Only return positive t values
    if (t1 < Ray.EPSILON) t1 = -1;
    if (t2 < Ray.EPSILON) t2 = -1;

    return new double[]{t1, t2};
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // Transform point to local space
    Point3 localPoint = inverseTransform.transformPoint(worldPoint);
}

```

```

// Determine which surface the hit point is on
// If the point is closer to the main sphere's surface (and outside the cut
sphere)
if (localPoint.distance(cutSphereCenter) >= cutRadius - Ray.EPSILON)
{ // Use distance()
    Vector3 normal = localPoint.subtract(mainSphereCenter).normalize();
    return
inverseTransposeTransform.transformVector(normal).normalize();
}
// If the point is on the cut sphere's surface (inside the main sphere)
else { // This implies localPoint.distance(cutSphereCenter) < cutRadius
- Ray.EPSILON
    // Normal for the cut surface should point inwards to form the crescent
    Vector3 normal = cutSphereCenter.subtract(localPoint).normalize(); //
Points towards cut sphere center
    return
inverseTransposeTransform.transformVector(normal).normalize();
}
}

```

@Override

```

public void setTransform(Matrix4 transform) {
    this.transform = transform;
    this.inverseTransform = transform.inverse();
    // For normals, we need the inverse transpose of the 3x3 part of the
transform matrix
    this.inverseTransposeTransform =
transform.inverseTransposeForNormal();
}

```

@Override

```

public Matrix4 getTransform() {
    return transform;
}

```

@Override

```

public Matrix4 getInverseTransform() {
    return inverseTransform;
}

```

```

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}
}
/**
// Create a thin crescent (hilal)
CrescentShape crescent = new CrescentShape(
2.0, // Main radius
1.8, // Cut radius
0.5 // Distance between centers
);

// Set material and transform as needed
crescent.setMaterial(new DiffuseMaterial(Color.YELLOW));
crescent.setTransform(Matrix4.rotationY(Math.PI/4));
*/

```

```

// =====
// File: /net/elenamurat/shape/EMShape.java
// =====

```

```

package net.elena.murat.shape;

import java.util.List;

//custom
import net.elena.murat.math.*;
import net.elena.murat.material.Material;

public interface EMShape {

```

```
//For CSG
List<IntersectionInterval> intersectAll(Ray ray);

//Old Methods
double intersect(Ray ray);

void setMaterial(Material material);
void setTransform(Matrix4 transform);

Vector3 getNormalAt(Point3 point);

Material getMaterial();

Matrix4 getTransform();
Matrix4 getInverseTransform();

}

// =====
// File: /net/elenamurat/shape/Cone.java
// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/**
 * Cone class represents a finite cone in 3D space, aligned along a local
axis.
 * Implements EMShape interface, now supporting Matrix4
transformations.
 */
public class Cone implements EMShape {
    private Material material;
```

```

// Cone's definition in its local space.
// Assuming a canonical cone: baseCenter at (0,0,0), axis along Y-axis
(0,1,0),
// and apex at (0, height, 0).
private final Point3 localBaseCenter;
private final double localRadius;
private final double localHeight;
private final Vector3 localAxis; // Canonical axis (0,1,0)
private final Point3 localApex; // Apex derived from localBaseCenter
and localHeight

// Transformation matrices
private Matrix4 transform; // Local to World transformation matrix
private Matrix4 inverseTransform; // World to Local transformation
matrix

/**
 * Constructs a cone with a base center, radius, and height in its LOCAL
coordinate system.
 * By default, the cone's base is at (0,0,0) and its axis extends along the
Y-axis.
 * @param radius The radius of the cone's base.
 * @param height The height of the cone.
 */
public Cone(double radius, double height) {
    // Define the cone in a canonical local space
    this.localBaseCenter = new Point3(0, 0, 0); // Base at origin
    this.localRadius = radius;
    this.localHeight = height;
    this.localAxis = new Vector3(0, 1, 0); // Axis along positive Y
    this.localApex =
localBaseCenter.add(this.localAxis.scale(localHeight)); // Apex at (0,
height, 0)

    // Initialize with identity transform by default
    this.transform = new Matrix4();
    this.inverseTransform = new Matrix4();
}

```

```
@Override  
public Material getMaterial() {  
    return material;  
}
```

```
@Override  
public void setMaterial(Material material) {  
    this.material = material;  
}
```

// --- New Methods for Transformation from EMShape interface ---

```
/**  
 * Sets the transformation matrix that converts points/vectors from the  
cone's  
 * local space to world space. When this is set, the inverse transform is  
also computed.  
 * @param transform The 4x4 transformation matrix.  
 */
```

```
@Override  
public void setTransform(Matrix4 transform) {  
    this.transform = transform;  
    this.inverseTransform = transform.inverse(); // Pre-compute inverse for  
efficiency  
}
```

```
/**  
 * Returns the transformation matrix that converts points/vectors from the  
cone's  
 * local space to world space.  
 * @return The 4x4 transformation matrix.  
 */
```

```
@Override  
public Matrix4 getTransform() {  
    return this.transform;  
}
```

```
/**  
 * Returns the inverse of the transformation matrix, which converts
```

points/vectors

- * from world space back to the cone's local space.
- * @return The 4x4 inverse transformation matrix.
- */

@Override

```
public Matrix4 getInverseTransform() {  
    return this.inverseTransform;  
}
```

/**

- * Calculates the normal vector at a given point on the cone's surface in WORLD coordinates.

- * The normal points outwards. This involves transforming the hit point to local space,

- * calculating the local normal, and then transforming it back to world space using the

- * inverse transpose of the model matrix.

- * @param worldPoint The point on the cone's surface in world coordinates.

- * @return The normalized normal vector at that point in world coordinates.

- */

@Override

```
public Vector3 getNormalAt(Point3 worldPoint) {  
    // 1. Transform the world hit point to local space  
    Point3 localHitPoint = inverseTransform.transformPoint(worldPoint);
```

Vector3 localNormal;

// Check if the hit point is on the base of the cone in local space

// localBaseCenter is (0,0,0) and localAxis is (0,1,0)

// So, base is at localHitPoint.y = 0

if (Math.abs(localHitPoint.y - localBaseCenter.y) < Ray.EPSILON) {

// Hit on the base

// The base normal is simply the negative local axis (pointing downwards)

localNormal = localAxis.negate();

} else {

// Hit on the lateral surface

```

// Vector from local apex to local hit point
Vector3 V = localHitPoint.subtract(localApex);

// Projection of V onto the local axis
double m = V.dot(localAxis);

// Radius-to-height ratio squared
double k_tan_sq = (localRadius * localRadius) / (localHeight * localHeight);

// Calculate local normal using the general cone normal formula
//  $N = (P_{\text{local}} - \text{Apex}_{\text{local}}) - ((P_{\text{local}} - \text{Apex}_{\text{local}}) \cdot \text{Axis}_{\text{local}}) * (1 + k_{\tan}^2) * \text{Axis}_{\text{local}}$ 
* (1 + k_tan_sq) * Axis_local
    // Simplified: Normal component perpendicular to axis is  $(P_{\text{local}} - \text{Proj}_{\text{on\_Axis}})$ 
        // The component along the axis is  $-k_{\tan}^2 * (\text{dist}_{\text{along\_axis\_from\_apex}}) * \text{Axis}$ 

    // Point on axis corresponding to the hit point's height
    Point3 projectionOnAxis = localApex.add(localAxis.scale(m / (1 + k_tan_sq))); // Corrected projection calculation for normal

    localNormal = localHitPoint.subtract(projectionOnAxis).normalize();

    // Ensure the normal points outwards. If the local axis is (0,1,0) and apex is above base,
        // the y-component of the normal should generally be positive or 0 for points near base
            // and negative for points on the side pointing "downwards" from apex's perspective.
                // A common heuristic is to check the dot product with a vector from apex to hit point
                    // or simply ensure it faces away from the cone's interior.
                    // For standard cone,  $(P_x, P_y, P_z)$  transformed to be  $(x,y,z)$  on cone centered at origin with apex on y axis:
                        //  $N = (x, -R/H * \sqrt{x^2+z^2}, z)$  for cone open downwards
                        // Or  $(x, R/H * \sqrt{x^2+z^2}, z)$  for cone open upwards.
                        // Our formula  $(P - \text{Proj})$  should usually give outward normal.
                        // Let's ensure it points "upwards" relative to the cone's center in local

```

space if it's pointing inwards.

```
// The provided formula  $N = V - (1 + k) * m * \text{axis}$  is for a cone whose  
axis points from base to apex.  
// Our localAxis points from base to apex. So,  $V = P - \text{Apex}$ .  
// A more robust normal calculation for cone lateral surface:  
// Vector from apex to current point:  $V_{\text{apex\_to\_point}} =$   
localHitPoint.subtract(localApex);  
// Height along axis: double current_height_along_axis =  
V_apex_to_point.dot(localAxis);  
// Radius at this height: double current_radius = localRadius * (1.0 -  
current_height_along_axis / localHeight);  
// Point on axis at this height: Point3 axis_point =  
localApex.add(localAxis.scale(current_height_along_axis));  
// Vector from axis to hit point: Vector3 radial_vec =  
localHitPoint.subtract(axis_point);  
// Normal = (radial_vec.normalize()).add(localAxis.scale(localRadius /  
localHeight)).normalize(); // This is for cone pointing upwards  
  
// Simpler calculation for Y-axis cone with apex at (0, height, 0) and  
base at (0,0,0):  
// The local normal points radially outwards and slightly  
upwards/downwards.  
// N_x = localHitPoint.x  
// N_z = localHitPoint.z  
// N_y = (localHitPoint.y - localApex.y) * (localRadius / localHeight) *  
(localRadius / localHeight) / localHitPoint.y.length()  
// No. It's:  
// Nx = (localHitPoint.x / localRadius) * localHeight  
// Ny = (localRadius / localHeight) * (localHitPoint.y - localApex.y) *  
-1.0  
// Nz = (localHitPoint.z / localRadius) * localHeight  
  
// A standard cone normal for a point (x,y,z) on the lateral surface of a  
cone with apex at (0,H,0) and base at (0,0,0)  
// is  $(x, -R/H * \sqrt{x^2+z^2}, z)$  normalized.  
// This simplifies to  $(x, -R/H * (\text{current\_radius at } y), z)$   
// Or, using the point-projection method:  
double r_sq_at_y = localHitPoint.subtract(localApex).lengthSquared()  
- Math.pow(localHitPoint.subtract(localApex).dot(localAxis), 2);
```

```

double current_radius = Math.sqrt(r_sq_at_y);

Vector3 vecFromApex = localHitPoint.subtract(localApex);
Vector3 radialDir =
vecFromApex.subtract(localAxis.scale(vecFromApex.dot(localAxis))).nor-
malize();

double cosAlpha = localHeight / Math.sqrt(localHeight * localHeight +
localRadius * localRadius);
double sinAlpha = localRadius / Math.sqrt(localHeight * localHeight +
localRadius * localRadius);

// Normal is composed of a radial component and an axial component
// For a cone pointing along +Y, radial component is horizontal.
// Axial component is typically upwards (negative if apex is above base
and normal points inwards)
localNormal = radialDir.add(localAxis.scale(cosAlpha /
sinAlpha)).normalize();

// If the cone is defined with apex at (0, height, 0) and base at (0,0,0),
// and localAxis (0,1,0), then the lateral normal's Y component should
be negative (pointing away from Y axis, "downwards")
// Ensure the normal points "outwards" relative to the Y-axis
projection:
if (localNormal.dot(localAxis) > 0) { // If normal points "upwards"
towards apex, flip it
    localNormal = localNormal.negate();
}
}

// 2. Transform the local normal back to world space
// Normals transform with the inverse transpose of the model matrix
//Matrix4 normalTransformMatrix =
this.inverseTransform.transpose(); // M_normal = (M^-1)^T
Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal(); // Normaller için
yeni metod
return
normalTransformMatrix.transformVector(localNormal).normalize();

```

```

}

/***
 * Finds the intersection of a ray with the cone in its local coordinate
system.
 * Returns the 't' value for the closest intersection point, or
Double.POSITIVE_INFINITY if no intersection.
 * The ray is first transformed into the cone's local coordinate system.
*/
@Override
public double intersect(Ray ray) {
    // 1. Transform the ray into the cone's local coordinate system
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
        inverseTransform.transformVector(ray.getDirection()).normalize();

    // Create a local ray
    Ray localRay = new Ray(localOrigin, localDirection);

    // Parameters for cone equation in canonical form (base at origin, apex
at (0, height, 0), axis along Y)
    //  $(x^2 + z^2) * H^2 = R^2 * (H - y)^2$ 
    // Or if apex is origin:  $x^2 + z^2 = (R/H * y)^2$  for y from 0 to H
    // Our cone has apex at (0, localHeight, 0) and base at (0,0,0).
    // The equation is  $x^2 + z^2 = (\text{localRadius} / \text{localHeight})^2 * (localHeight - y)^2$ 
    // Let k =  $(\text{localRadius} / \text{localHeight})^2$ 
    //  $x^2 + z^2 - k * (localHeight - y)^2 = 0$ 

    // Ray in local space:  $P(t) = O_{\text{local}} + t * D_{\text{local}}$ 
    //  $P_x = O_x + t*D_x$ 
    //  $P_y = O_y + t*D_y$ 
    //  $P_z = O_z + t*D_z$ 

    // Substitute into cone equation:
    //  $(O_x + t*D_x)^2 + (O_z + t*D_z)^2 - k * (localHeight - (O_y + t*D_y))^2 = 0$ 
    //  $(O_x + t*D_x)^2 + (O_z + t*D_z)^2 - k * ((localHeight - O_y) - t*D_y)^2 = 0$ 
}

```

```

double k_cone = (localRadius * localRadius) / (localHeight *
localHeight);

// Coefficients for quadratic equation At^2 + Bt + C = 0
// A = D_x^2 + D_z^2 - k * D_y^2
double a = (localRay.getDirection().x * localRay.getDirection().x) +
(localRay.getDirection().z * localRay.getDirection().z) -
k_cone * (localRay.getDirection().y * localRay.getDirection().y);

// B = 2 * (O_x*D_x + O_z*D_z - k * D_y * (O_y - localHeight))
double b = 2 * ((localRay.getOrigin().x * localRay.getDirection().x) +
(localRay.getOrigin().z * localRay.getDirection().z) -
k_cone * localRay.getDirection().y * (localRay.getOrigin().y -
localHeight));

// C = O_x^2 + O_z^2 - k * (O_y - localHeight)^2
double c = (localRay.getOrigin().x * localRay.getOrigin().x) +
(localRay.getOrigin().z * localRay.getOrigin().z) -
k_cone * Math.pow(localRay.getOrigin().y - localHeight, 2);

double discriminant = b * b - 4 * a * c;

double tLateral = Double.POSITIVE_INFINITY; // Intersection with
lateral surface

if (discriminant >= Ray.EPSILON) {
    double sqrtD = Math.sqrt(discriminant);
    double t0 = (-b - sqrtD) / (2 * a);
    double t1 = (-b + sqrtD) / (2 * a);

    if (t0 > t1) { // Ensure t0 is the smaller positive root
        double temp = t0;
        t0 = t1;
        t1 = temp;
    }

    // Check if t0 is a valid intersection on the lateral surface
    if (t0 > Ray.EPSILON) {

```

```

Point3 hitPoint0 = localRay.pointAtParameter(t0);
// Check if hitPoint0 is within the cone's height bounds [0,
localHeight]
boolean isWithinHeight0 = (hitPoint0.y <= (localHeight +
Ray.EPSILON)) && (hitPoint0.y >= (localBaseCenter.y -
Ray.EPSILON));
if (isWithinHeight0) {
    tLateral = t0;
}
}

// If t0 was not valid or t1 is closer and valid, check t1
if (t1 > Ray.EPSILON && t1 < tLateral) { // Only check t1 if it's
potentially closer than t0 or if t0 was invalid
    Point3 hitPoint1 = localRay.pointAtParameter(t1);
    boolean isWithinHeight1 = (hitPoint1.y <= (localHeight +
Ray.EPSILON)) && (hitPoint1.y >= (localBaseCenter.y -
Ray.EPSILON));
    if (isWithinHeight1) {
        tLateral = t1;
    }
}

// --- Base Intersection Calculation (in local space) ---
double tBase = Double.POSITIVE_INFINITY;
// The base is a circle on the y=0 plane (localBaseCenter.y)
// Normal of the base is (0, -1, 0) since localAxis is (0,1,0) and base is
below apex.
Vector3 baseNormal = localAxis.negate(); // Points downwards from
base

double denomBase = localRay.getDirection().dot(baseNormal);

if (Math.abs(denomBase) > Ray.EPSILON) { // Ray is not parallel to
the base plane
    //  $t = (\mathbf{P}_{\text{base}} - \mathbf{O}_{\text{ray\_local}}) \cdot \mathbf{N}_{\text{base}} / (\mathbf{D}_{\text{ray\_local}} \cdot \mathbf{N}_{\text{base}})$ 
    tBase =
(localBaseCenter.subtract(localRay.getOrigin())).dot(baseNormal) /

```

```

denomBase;

if (tBase > Ray.EPSILON) { // Intersection is in front of the ray
    Point3 hitPointBase = localRay.pointAtParameter(tBase);
    // Check if the intersection point is within the base circle's radius
    double distSqFromBaseCenter =
        hitPointBase.subtract(localBaseCenter).lengthSquared();
    if (distSqFromBaseCenter <= localRadius * localRadius +
        Ray.EPSILON) {
        // Valid base intersection
    } else {
        tBase = Double.POSITIVE_INFINITY; // Outside base circle
    }
} else {
    tBase = Double.POSITIVE_INFINITY; // Intersection is behind the
ray's origin
}
}

// Return the closest valid intersection (between lateral surface and base)
double finalT = Math.min(tLateral, tBase);
return finalT == Double.POSITIVE_INFINITY ? -1 : finalT; // Return -1
if no intersection
}

/***
 * Calculates all intersection intervals between a ray and this finite cone.
 * The ray is transformed into the cone's local space for calculation.
 * The method checks for intersections with the lateral (conical) surface
and the base.
 * All valid intersections are collected, sorted by t, and paired into
intervals.
 * @param ray The ray to test, in world coordinates.
 * @return A list of IntersectionInterval objects. Empty if no
intersections.
 */
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into the cone's local coordinate system
}

```

```

Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
Vector3 localDirection =
inverseTransform.transformVector(ray.getDirection()).normalize();
Ray localRay = new Ray(localOrigin, localDirection);

// List to store all intersection points
java.util.List<Intersection> hits = new java.util.ArrayList<>();

// 2. Lateral (conical) surface intersection
double k_cone = (localRadius * localRadius) / (localHeight *
localHeight);
double dx = localDirection.x, dy = localDirection.y, dz =
localDirection.z;
double ox = localOrigin.x, oy = localOrigin.y, oz = localOrigin.z;

// Coefficients for quadratic equation: At^2 + Bt + C = 0
double a = dx*dx + dz*dz - k_cone*dy*dy;
double b = 2 * (ox*dx + oz*dz - k_cone*dy*(oy - localHeight));
double c = ox*ox + oz*oz - k_cone*(oy - localHeight)*(oy -
localHeight);

if (Math.abs(a) > Ray.EPSILON) {
    double discriminant = b*b - 4*a*c;
    if (discriminant >= 0) {
        double sqrtD = Math.sqrt(discriminant);
        double t0 = (-b - sqrtD) / (2*a);
        double t1 = (-b + sqrtD) / (2*a);

        // Check t0
        if (t0 > Ray.EPSILON) {
            Point3 hit = localRay.pointAtParameter(t0);
            if (hit.y >= -Ray.EPSILON && hit.y <= localHeight +
Ray.EPSILON) {
                Point3 worldHit = transform.transformPoint(hit);
                Vector3 worldNormal = getNormalAt(worldHit);
                hits.add(new Intersection(worldHit, worldNormal, t0, this));
            }
        }
    }
}

```

```

// Check t1
if (t1 > Ray.EPSILON) {
    Point3 hit = localRay.pointAtParameter(t1);
    if (hit.y >= -Ray.EPSILON && hit.y <= localHeight +
Ray.EPSILON) {
        Point3 worldHit = transform.transformPoint(hit);
        Vector3 worldNormal = getNormalAt(worldHit);
        hits.add(new Intersection(worldHit, worldNormal, t1, this));
    }
}
}

// 3. Base intersection (circle at y=0 in local space)
Vector3 baseNormalLocal = localAxis.negate(); // (0, -1, 0)
double denom = localDirection.dot(baseNormalLocal);
if (Math.abs(denom) > Ray.EPSILON) {
    double t =
(localBaseCenter.subtract(localOrigin)).dot(baseNormalLocal) / denom;
    if (t > Ray.EPSILON) {
        Point3 localHit = localRay.pointAtParameter(t);
        double distSq = localHit.subtract(localBaseCenter).lengthSquared();
        if (distSq <= localRadius * localRadius + Ray.EPSILON) {
            Point3 worldHit = transform.transformPoint(localHit);
            Vector3 worldNormal = getNormalAt(worldHit);
            hits.add(new Intersection(worldHit, worldNormal, t, this));
        }
    }
}
}

// 4. Sort all intersections by t
java.util.Collections.sort(hits, (i1, i2) -> Double.compare(i1.getT(),
i2.getT()));

// 5. Pair into intervals
List<IntersectionInterval> intervals = new java.util.ArrayList<>();
for (int i = 0; i < hits.size() - 1; i += 2) {
    Intersection in = hits.get(i);
    Intersection out = hits.get(i + 1);
}

```

```

        intervals.add(new IntersectionInterval(in.getT(), out.getT(), in, out));
    }

    return intervals;
}

}

// =====
// File: /net/elenamurat/shape/Torus.java
// =====

package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

public class Torus implements EMShape {
    // Definition of torus in local coordinate system
    // Canonical Torus: centered at (0,0,0), aligned with Y axis
    private final double majorRadius; // R
    private final double minorRadius; // r
    private Material material;

    // Transformation matrices for EMShape interface
    private Matrix4 transform;      // From local space to world space
    private Matrix4 inverseTransform; // From world space to local space
    (inverse of transform)

    // Ray Marching Parameters
    private static final int MAX_MARCH_STEPS = 200;
    private static final double HIT_THRESHOLD = 0.001; // Threshold for
distance function
    private static final double MAX_MARCH_DISTANCE = 100.0; //
Maximum marching distance
}

```

```

/***
 * Creates a Torus defined in local coordinate system.
 * Centered at (0,0,0). World space position and transformations are set
via setTransform().
 * @param majorRadius Main radius of torus (from center to tube center)
 * @param minorRadius Secondary radius of torus (tube thickness)
*/
public Torus(double majorRadius, double minorRadius) {
    this.majorRadius = majorRadius;
    this.minorRadius = minorRadius;
    this.material = null; // Material initially null

    // Initialize with identity transformation matrices by default
    this.transform = new Matrix4();
    this.inverseTransform = new Matrix4();
}

@Override
public Material getMaterial() {
    return material;
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

// --- EMShape Interface Implementations ---

/***
 * Sets the transformation matrix from object's local space to world space.
 * When this matrix is set, the inverse transformation matrix is also
calculated.
 * @param transform 4x4 transformation matrix (may contain translation,
rotation, scaling).
*/
@Override
public void setTransform(Matrix4 transform) {
    this.transform = transform;
}

```

```

    this.inverseTransform = transform.inverse(); // Pre-calculate inverse for
efficiency

    if (this.inverseTransform == null) {
        System.err.println("Warning: Could not compute inverse transform for
Torus (determinant zero). Using identity matrix.");
        this.inverseTransform = new Matrix4();
    }
}

/**
 * Returns the transformation matrix from object's local space to world
space.
 * @return 4x4 transformation matrix.
 */
@Override
public Matrix4 getTransform() {
    return this.transform;
}

/**
 * Returns the transformation matrix from world space to object's local
space.
 * @return 4x4 inverse transformation matrix.
 */
@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}

/**
 * Calculates the SDF distance of a given point in the torus's local
coordinate system.
 * This method assumes the torus is centered at origin and aligned with Y
axis.
 * @param p Point in object's local space.
 * @return SDF distance from point to torus.
 */
private double calculateSDFLocal(Point3 p) {

```

```

// q.x gives distance to torus's main circle (in XZ plane)
// q.y gives distance along torus's Z axis (donut along Y axis)
// Note: Previous code used Z as Y, which is a common approach in
SDFs
    // (x, y, z) -> (x, z, y) or (sqrt(x^2+y^2)-R, z)
    // If torus's main circle is in XY plane:
    // double q_x = new Vector3(p.x, p.y, 0).length() - majorRadius; //
sqrt(x^2 + y^2) - R
    // double q_y = p.z; // Z component

    // Following previous code's logic (majorRadius in XY plane,
minorRadius in Z direction)
    // So, Torus rotates around Y axis and its main plane is XZ plane.
    double q_x_dist = new Vector3(p.x, p.z, 0).length() - majorRadius; //
Circular distance in XZ plane - major radius
    double q_y_val = p.y; // Torus's own "height" axis (minorRadius
direction)

    return Math.sqrt(q_x_dist * q_x_dist + q_y_val * q_y_val) -
minorRadius;
}

/***
 * Checks if a ray intersects this Torus object using Ray Marching
technique.
 *
 * @param ray Ray to test for intersection (in world space).
 * @return Distance from ray origin to intersection point (t) if exists,
otherwise Double.POSITIVE_INFINITY.
 */
@Override
public double intersect(Ray ray) {
    // 1. Transform ray into object's local coordinate system
    Point3 localRayOrigin =
inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localRayDirection =
inverseTransform.transformVector(ray.getDirection()).normalize();

    double totalDistanceMarched = 0.0;

```

```

Point3 currentLocalRayPosition = localRayOrigin; // Ray's current
position (in local space)

for (int i = 0; i < MAX_MARCH_STEPS; i++) {
    // Calculate distance from current position to torus (in local space)
    double distanceToTorus =
calculateSDFLocal(currentLocalRayPosition);

    // If distance is below threshold, we found an intersection
    if (distanceToTorus < HIT_THRESHOLD) {
        return totalDistanceMarched; // Return total marched distance
    }

    // Advance ray by distance to object (in local space)
    currentLocalRayPosition =
currentLocalRayPosition.add(localRayDirection.scale(distanceToTorus));
    totalDistanceMarched += distanceToTorus;

    // If total marched distance exceeds maximum distance, no intersection
    if (totalDistanceMarched >= MAX_MARCH_DISTANCE) {
        return Double.POSITIVE_INFINITY; // No intersection
    }
}

// If maximum steps reached without finding intersection
return Double.POSITIVE_INFINITY;
}

/***
 * Calculates all intersection intervals between a ray and this torus using
ray marching.
 * Detects both entry (tIn) and exit (tOut) points by monitoring the SDF
sign change.
 * @param ray The ray to test, in world coordinates.
 * @return A list of IntersectionInterval objects. Empty if no valid
interval.
 */
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into local space

```

```

Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
Vector3 localDirection =
inverseTransform.transformVector(ray.getDirection()).normalize();

double totalDistance = 0.0;
Point3 currentPosition = localOrigin;
List<IntersectionInterval> intervals = new java.util.ArrayList<>();

// Track if we are currently inside the shape
boolean isInside = false;
double tIn = -1;

for (int i = 0; i < MAX_MARCH_STEPS; i++) {
    double sdf = calculateSDFLocal(currentPosition);

    boolean wasInside = isInside;
    isInside = sdf < HIT_THRESHOLD;

    // Entry: from outside to inside
    if (!wasInside && isInside && tIn < 0) {
        tIn = totalDistance;
    }

    // Exit: from inside to outside
    if (wasInside && !isInside && tIn >= 0) {
        double tOut = totalDistance;

        // Create Intersection objects
        Point3 pointIn = ray.pointAtParameter(tIn);
        Point3 pointOut = ray.pointAtParameter(tOut);
        Vector3 normalIn = getNormalAt(pointIn);
        Vector3 normalOut = getNormalAt(pointOut);
        Intersection in = new Intersection(pointIn, normalIn, tIn, this);
        Intersection out = new Intersection(pointOut, normalOut, tOut, this);

        intervals.add(new IntersectionInterval(tIn, tOut, in, out));
        tIn = -1; // Reset for next interval
    }
}

```

```

// Move ray forward
double step = Math.max(sdf, 0.01); // Avoid zero step
currentPosition = currentPosition.add(localDirection.scale(step));
totalDistance += step;

if (totalDistance > MAX_MARCH_DISTANCE) {
    break;
}
}

return intervals;
}

/**
 * Calculates the surface normal at intersection point.
 * Normal is approximated using gradient of SDF function (via finite
differences method).
*
* @param worldPoint Intersection point (in world space).
* @return Normalized surface normal at intersection point (in world
space).
*/
@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    // 1. Transform intersection point to object's local space
    Point3 localIntersectionPoint =
inverseTransform.transformPoint(worldPoint);

    final double h = 0.0001; // Small perturbation (epsilon) for gradient
calculation

    // Calculate normal in local space (numerical differentiation -
approximate gradient)
    // Using SDF value changes with dx, dy, dz increments
    double nx = calculateSDFLocal(localIntersectionPoint.add(new
Vector3(h, 0, 0))) - calculateSDFLocal(localIntersectionPoint.add(new
Vector3(-h, 0, 0)));
    double ny = calculateSDFLocal(localIntersectionPoint.add(new
Vector3(0, h, 0))) - calculateSDFLocal(localIntersectionPoint.add(new

```

```

Vector3(0, -h, 0));
    double nz = calculateSDFLocal(localIntersectionPoint.add(new
Vector3(0, 0, h))) - calculateSDFLocal(localIntersectionPoint.add(new
Vector3(0, 0, -h)));
}

Vector3 localNormal = new Vector3(nx, ny, nz).normalize();

// 2. Transform local normal back to world space
// Normals are transformed using inverse transpose of model matrix
Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal(); // New method for
normals
return
normalTransformMatrix.transformVector(localNormal).normalize(); ///
Normalize after transformation
}

```

```

// =====
// File: /net/elenamurat/shape/TransparentPlane.java
// =====

```

```

package net.elena.murat.shape;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

import java.util.List;

public class TransparentPlane implements EMShape {
    private final double width, height, thickness;
    private Material material;
    private Matrix4 transform = Matrix4.identity();
    private Matrix4 inverseTransform = Matrix4.identity();

    private static final Vector3 VEC_Y = new Vector3(0, 1, 0);
    private static final Vector3 VEC_X = new Vector3(1, 0, 0);

```

```
public TransparentPlane(Point3 pointOnPlane, Vector3 normal, double thickness) {
    this.thickness = thickness;
    this.width = 2000.0;
    this.height = 2000.0;

    normal = normal.normalize();

    // Basis vectors
    Vector3 up = Math.abs(normal.dot(VEC_Y)) > 0.99 ? VEC_X :
    VEC_Y;
    Vector3 right = normal.cross(up).normalize();
    Vector3 forward = right.cross(normal).normalize();

    // Build transformation matrix: rotation + translation
    Matrix4 rotTrans = new Matrix4(
        right.x,   forward.x,   normal.x,   pointOnPlane.x,
        right.y,   forward.y,   normal.y,   pointOnPlane.y,
        right.z,   forward.z,   normal.z,   pointOnPlane.z,
        0,         0,           0,           1
    );
    this.setTransform(rotTrans);
}
```

```
@Override
public void setMaterial(Material material) {
    this.material = material;
}
```

```
@Override
public Material getMaterial() {
    return this.material;
}
```

```
@Override
public void setTransform(Matrix4 transform) {
    this.transform = transform;
```

```
this.inverseTransform = transform.inverse();
if (this.inverseTransform == null) {
    System.err.println("TransparentPlane: Non-invertible transform!");
    this.inverseTransform = Matrix4.identity();
}
}
```

```
@Override
public Matrix4 getTransform() {
    return this.transform;
}
```

```
@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}
```

```
@Override
public double intersect(Ray ray) {
    // Transform ray to local space
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDir =
    inverseTransform.transformVector(ray.getDirection()).normalize();
    Ray localRay = new Ray(localOrigin, localDir);

    // Local box bounds
    double w = width / 2.0;
    double h = height / 2.0;
    double t = thickness / 2.0;
```

```
// Slab method
double tmin = -Double.MAX_VALUE;
double tmax = Double.MAX_VALUE;

// X
if (Math.abs(localRay.getDirection().x) > Ray.EPSILON) {
    double tx1 = (-w - localRay.getOrigin().x) / localRay.getDirection().x;
    double tx2 = (w - localRay.getOrigin().x) / localRay.getDirection().x;
    tmin = Math.max(tmin, Math.min(tx1, tx2));
```

```

tmax = Math.min(tmax, Math.max(tx1, tx2));
} else if (localRay.getOrigin().x < -w || localRay.getOrigin().x > w) {
    return -1;
}

// Y
if (Math.abs(localRay.getDirection().y) > Ray.EPSILON) {
    double ty1 = (-h - localRay.getOrigin().y) / localRay.getDirection().y;
    double ty2 = (h - localRay.getOrigin().y) / localRay.getDirection().y;
    tmin = Math.max(tmin, Math.min(ty1, ty2));
    tmax = Math.min(tmax, Math.max(ty1, ty2));
} else if (localRay.getOrigin().y < -h || localRay.getOrigin().y > h) {
    return -1;
}

// Z
if (Math.abs(localRay.getDirection().z) > Ray.EPSILON) {
    double tz1 = (-t - localRay.getOrigin().z) / localRay.getDirection().z;
    double tz2 = (t - localRay.getOrigin().z) / localRay.getDirection().z;
    tmin = Math.max(tmin, Math.min(tz1, tz2));
    tmax = Math.min(tmax, Math.max(tz1, tz2));
} else if (localRay.getOrigin().z < -t || localRay.getOrigin().z > t) {
    return -1;
}

if (tmax >= tmin && tmax > Ray.EPSILON) {
    return tmin > Ray.EPSILON ? tmin : tmax;
}
return -1;
}

@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    double t = intersect(ray);
    if (t <= Ray.EPSILON) return java.util.Collections.emptyList();

    Point3 hit = ray.pointAtParameter(t);
    Vector3 norm = getNormalAt(hit);
    Intersection inter = new Intersection(hit, norm, t, this);
}

```

```
    return java.util.Arrays.asList(IntersectionInterval.point(t, inter));
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    Point3 local = inverseTransform.transformPoint(worldPoint);
    double w = width / 2.0;
    double h = height / 2.0;
    double t = thickness / 2.0;

    if (Math.abs(local.x) > w - Ray.EPSILON) {
        return transform.transformVector(new Vector3(local.x > 0 ? 1 : -1, 0,
0)).normalize();
    } else if (Math.abs(local.y) > h - Ray.EPSILON) {
        return transform.transformVector(new Vector3(0, local.y > 0 ? 1 : -1,
0)).normalize();
    } else if (Math.abs(local.z) > t - Ray.EPSILON) {
        return transform.transformVector(new Vector3(0, 0, local.z > 0 ? 1 : -1)).normalize();
    }

    return transform.transformVector(new Vector3(0, 0, 1)).normalize();
}
```

```
// =====
// File: /net/elenamurat/shape/RectangularPrism.java
// =====
```

```
package net.elena.murat.shape;

import java.util.List;

// Custom imports
import net.elena.murat.material.Material;
import net.elena.murat.math.*;
```

```

/***
 * Represents an axis-aligned rectangular prism (or cuboid) in 3D space.
 * It implements EMShape to support transformations and materials.
 * The prism is defined by its width, height, and depth, centered at its local
origin (0,0,0).
 */
public class RectangularPrism implements EMShape {

    private final double width;
    private final double height;
    private final double depth;

    // EMShape interface transformation matrices and material
    private Matrix4 transform;
    private Matrix4 inverseTransform;
    private Matrix4 inverseTransposeTransformForNormal; // For correct
normal transformation
    private Material material;

    /**
     * Constructs a RectangularPrism with specified dimensions.
     * The prism is initially axis-aligned and centered at (0,0,0) in its local
space.
     *
     * @param width The width of the prism along the local X-axis.
     * @param height The height of the prism along the local Y-axis.
     * @param depth The depth of the prism along the local Z-axis.
     */
    public RectangularPrism(double width, double height, double depth) {
        this.width = width;
        this.height = height;
        this.depth = depth;
        // Default transform is identity.
        this.transform = Matrix4.identity();
        updateTransforms(); // Inverse transforms are calculated initially
    }

    /**
     * Constructs a RectangularPrism with specified dimensions and a

```

material.

*

* @param width The width of the prism along the local X-axis.
* @param height The height of the prism along the local Y-axis.
* @param depth The depth of the prism along the local Z-axis.
* @param material The material applied to the prism.

*/

```
public RectangularPrism(double width, double height, double depth,  
Material material) {  
    this(width, height, depth);  
    this.setMaterial(material);  
}
```

// --- EMShape Interface Methods Implementation ---

@Override

```
public void setTransform(Matrix4 transform) {  
    // Create a deep copy of the incoming matrix to prevent external  
    modifications  
    this.transform = new Matrix4(transform);  
    updateTransforms(); // Update inverse matrices when transform changes  
}
```

@Override

```
public Matrix4 getTransform() {  
    return this.transform;  
}
```

@Override

```
public Matrix4 getInverseTransform() {  
    return this.inverseTransform;  
}
```

/**

* Updates the inverse and inverse transpose transforms whenever the
main transform changes.

* This method is called internally by setTransform and the constructor.
*/

```
private void updateTransforms() {
```

```

this.inverseTransform = this.transform.inverse();
this.inverseTransposeTransformForNormal =
this.transform.inverseTransposeForNormal();
}

@Override
public void setMaterial(Material material) {
    this.material = material;
}

@Override
public Material getMaterial() {
    return this.material;
}

/**
 * Calculates the intersection of a ray with the rectangular prism.
 * This method transforms the ray into the object's local space,
 * performs the intersection test, and returns the 't' value.
 *
 * @param ray The ray to intersect with the prism (in world coordinates).
 * @return The distance 't' along the ray to the closest intersection point,
 * or Double.POSITIVE_INFINITY if no intersection.
 */
@Override
public double intersect(Ray ray) {
    // Ensure transforms are not null before proceeding
    if (inverseTransform == null || inverseTransposeTransformForNormal
    == null) {
        System.err.println("Error: RectangularPrism transforms are null.
Cannot intersect.");
        return Double.POSITIVE_INFINITY; // Return infinity if transforms
        are invalid
    }

    // 1. Transform the ray into the prism's local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize() //

```

Normalize direction after transformation

);

```
double tMin = Double.NEGATIVE_INFINITY;  
double tMax = Double.POSITIVE_INFINITY;
```

```
// Calculate half dimensions for easier calculations  
double halfWidth = width / 2.0;  
double halfHeight = height / 2.0;  
double halfDepth = depth / 2.0;
```

```
// Intersection with X-planes (slab method)
```

```
// Handle cases where ray direction component is zero to avoid division  
by zero
```

```
if (Math.abs(localRay.getDirection().x) < Ray.EPSILON) {  
    if (localRay.getOrigin().x < -halfWidth || localRay.getOrigin().x >  
halfWidth) {  
        return Double.POSITIVE_INFINITY; // Ray is parallel and outside  
the slab
```

```
}
```

```
} else {
```

```
    double t1 = (-halfWidth - localRay.getOrigin().x) /  
localRay.getDirection().x;
```

```
    double t2 = (halfWidth - localRay.getOrigin().x) /  
localRay.getDirection().x;
```

```
    if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; } // Ensure t1 is min,  
t2 is max
```

```
    tMin = Math.max(tMin, t1);
```

```
    tMax = Math.min(tMax, t2);
```

```
    if (tMin > tMax) return Double.POSITIVE_INFINITY; // No  
intersection
```

```
}
```

```
// Intersection with Y-planes (slab method)
```

```
if (Math.abs(localRay.getDirection().y) < Ray.EPSILON) {  
    if (localRay.getOrigin().y < -halfHeight || localRay.getOrigin().y >  
halfHeight) {  
        return Double.POSITIVE_INFINITY;  
    }
```

```

    } else {
        double t1 = (-halfHeight - localRay.getOrigin().y) /
localRay.getDirection().y;
        double t2 = (halfHeight - localRay.getOrigin().y) /
localRay.getDirection().y;
        if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return Double.POSITIVE_INFINITY;
    }

    // Intersection with Z-planes (slab method)
    if (Math.abs(localRay.getDirection().z) < Ray.EPSILON) {
        if (localRay.getOrigin().z < -halfDepth || localRay.getOrigin().z >
halfDepth) {
            return Double.POSITIVE_INFINITY;
        }
    } else {
        double t1 = (-halfDepth - localRay.getOrigin().z) /
localRay.getDirection().z;
        double t2 = (halfDepth - localRay.getOrigin().z) /
localRay.getDirection().z;
        if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return Double.POSITIVE_INFINITY;
    }

    double t = tMin;
    if (t < Ray.EPSILON) { // If closest intersection is behind or too close to
origin
        t = tMax; // Try the farther intersection
        if (t < Ray.EPSILON) { // If farther intersection is also behind
            return Double.POSITIVE_INFINITY; // No valid intersection
        }
    }

    return t; // Return the valid intersection distance
}

```

```

/**
 * Calculates all intersection intervals between a ray and this rectangular
prism.
 * Uses the slab method to find entry and exit points on the six faces.
 * @param ray The ray to test, in world coordinates.
 * @return A list of IntersectionInterval objects. Empty if no intersection.
*/
@Override
public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Check for valid transforms
    if (inverseTransform == null || inverseTransposeTransformForNormal
== null) {
        return java.util.Collections.emptyList();
    }

    // 2. Transform the ray into local space
    Ray localRay = new Ray(
        inverseTransform.transformPoint(ray.getOrigin()),
        inverseTransform.transformVector(ray.getDirection()).normalize()
    );

    double tMin = Double.NEGATIVE_INFINITY;
    double tMax = Double.POSITIVE_INFINITY;

    double halfWidth = width / 2.0;
    double halfHeight = height / 2.0;
    double halfDepth = depth / 2.0;

    // Slab intersection on X, Y, Z axes
    double[][][] slabs = {
        { -halfWidth, halfWidth, localRay.getDirection().x,
localRay.getOrigin().x },
        { -halfHeight, halfHeight, localRay.getDirection().y,
localRay.getOrigin().y },
        { -halfDepth, halfDepth, localRay.getDirection().z,
localRay.getOrigin().z }
    };
}

```

```

for (double[] slab : slabs) {
    double minBound = slab[0], maxBound = slab[1];
    double dir = slab[2], origin = slab[3];

    if (Math.abs(dir) < Ray.EPSILON) {
        if (origin < minBound || origin > maxBound) {
            return java.util.Collections.emptyList();
        }
    } else {
        double t1 = (minBound - origin) / dir;
        double t2 = (maxBound - origin) / dir;
        if (t1 > t2) { double temp = t1; t1 = t2; t2 = temp; }
        tMin = Math.max(tMin, t1);
        tMax = Math.min(tMax, t2);
        if (tMin > tMax) return java.util.Collections.emptyList();
    }
}

// 3. Now we have tMin (entry) and tMax (exit)
if (tMax < Ray.EPSILON) return java.util.Collections.emptyList();
if (tMin < Ray.EPSILON) tMin = tMax; // Ray inside the box

// 4. Create Intersection objects
Point3 pointIn = ray.pointAtParameter(tMin);
Vector3 normalIn = getNormalAt(pointIn);
Intersection in = new Intersection(pointIn, normalIn, tMin, this);

Point3 pointOut = ray.pointAtParameter(tMax);
Vector3 normalOut = getNormalAt(pointOut);
Intersection out = new Intersection(pointOut, normalOut, tMax, this);

// 5. Return the interval
return java.util.Arrays.asList(new IntersectionInterval(tMin, tMax, in,
out));
}

/**
 * Returns the surface normal at a given point on the prism's surface.
 * The point is in world coordinates. The normal is calculated in local

```

```
space
```

```
* and then transformed back to world space.
```

```
*
```

```
* @param worldPoint The point on the prism's surface in world  
coordinates.
```

```
* @return The normalized normal vector at that point in world  
coordinates.
```

```
*/
```

```
@Override
```

```
public Vector3 getNormalAt(Point3 worldPoint) {
```

```
    // Ensure transforms are not null before proceeding
```

```
    if (inverseTransform == null || inverseTransposeTransformForNormal  
== null) {
```

```
        System.err.println("Error: RectangularPrism transforms are null.  
Cannot compute normal.");
```

```
        // Fallback: return a default normal or throw an exception
```

```
        return new Vector3(0, 1, 0);
```

```
}
```

```
// Transform the world point to the prism's local space
```

```
Point3 localPoint =
```

```
this.getInverseTransform().transformPoint(worldPoint);
```

```
Vector3 localNormal;
```

```
// Use a slightly larger epsilon for normal calculation to avoid ambiguity  
at edges/corners
```

```
double normalEpsilon = Ray.EPSILON * 10;
```

```
double halfWidth = width / 2.0;
```

```
double halfHeight = height / 2.0;
```

```
double halfDepth = depth / 2.0;
```

```
// Determine which face the local point is on by checking which  
coordinate is closest to the boundary
```

```
if (Math.abs(localPoint.x - halfWidth) < normalEpsilon) {
```

```
    localNormal = new Vector3(1, 0, 0);
```

```
} else if (Math.abs(localPoint.x + halfWidth) < normalEpsilon) {
```

```
    localNormal = new Vector3(-1, 0, 0);
```

```

}

else if (Math.abs(localPoint.y - halfHeight) < normalEpsilon) {
    localNormal = new Vector3(0, 1, 0);
} else if (Math.abs(localPoint.y + halfHeight) < normalEpsilon) {
    localNormal = new Vector3(0, -1, 0);
}
else if (Math.abs(localPoint.z - halfDepth) < normalEpsilon) {
    localNormal = new Vector3(0, 0, 1);
} else if (Math.abs(localPoint.z + halfDepth) < normalEpsilon) {
    localNormal = new Vector3(0, 0, -1);
} else {
    // Fallback for floating point inaccuracies near edges/corners.
    // This attempts to find the closest face based on the local point's
coordinates.

    double[] dists = {
        Math.abs(localPoint.x - halfWidth),
        Math.abs(localPoint.x + halfWidth),
        Math.abs(localPoint.y - halfHeight),
        Math.abs(localPoint.y + halfHeight),
        Math.abs(localPoint.z - halfDepth),
        Math.abs(localPoint.z + halfDepth)
    };
    int minIdx = 0;
    for(int i = 1; i < 6; i++) {
        if (dists[i] < dists[minIdx]) {
            minIdx = i;
        }
    }
    switch(minIdx) {
        case 0: localNormal = new Vector3(1, 0, 0); break;
        case 1: localNormal = new Vector3(-1, 0, 0); break;
        case 2: localNormal = new Vector3(0, 1, 0); break;
        case 3: localNormal = new Vector3(0, -1, 0); break;
        case 4: localNormal = new Vector3(0, 0, 1); break;
        case 5: localNormal = new Vector3(0, 0, -1); break;
        default: localNormal = new Vector3(0, 1, 0); // Should not happen
    }
    System.err.println("Warning: RectangularPrism normal fallback used
due to floating point inaccuracy.");
}

```

```
    }

    // Transform the local normal back to world space
    // Use inverse transpose for correct normal transformation
    return
this.inverseTransposeTransformForNormal.transformVector(localNormal)
.normalize();
}

}
```

```
// =====
// File: /net/elenamurat/shape/Cylinder.java
// =====
```

```
package net.elena.murat.shape;

import java.util.List;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;

/***
 * Finite cylinder aligned along the Y-axis in its local coordinate system.
 * Implements EMShape interface, now supporting Matrix4
transformations.
 */
public class Cylinder implements EMShape {
    private Material material;

    // Cylinder's definition in its LOCAL coordinate system.
    // Canonical cylinder: base center at (0,0,0), radius 'localRadius', height
    'localHeight',
    // aligned along the positive Y-axis (localAxis = 0,1,0).
    private final Point3 localBaseCenter;
    private final double localRadius;
    private final double localHeight;
    private final Vector3 localAxis; // Always (0,1,0) in local space
```

```

// Transformation matrices
private Matrix4 transform;      // Local to World transformation matrix
private Matrix4 inverseTransform; // World to Local transformation
matrix

// Using a class-level EPSILON for consistency
private static final double EPSILON = 1e-5;

/***
 * Constructs a cylinder with a given radius and height.
 * The cylinder's base is implicitly at (0,0,0) in its LOCAL coordinate
system,
 * and it extends along the positive Y-axis up to (0, height, 0).
 * @param radius The radius of the cylinder.
 * @param height The height of the cylinder.
 */
public Cylinder(double radius, double height) {
    this.localBaseCenter = new Point3(0, 0, 0);
    this.localRadius = radius;
    this.localHeight = height;
    this.localAxis = new Vector3(0, 1, 0);
    this.transform = new Matrix4();
    this.inverseTransform = new Matrix4();
}

/***
 * NEW CONSTRUCTOR - Creates a cylinder between two points in
world space
*/
public Cylinder(Point3 startPoint, Point3 endPoint, double radius, double
height) {
    this(radius, height); // Call basic constructor first

    Vector3 direction = endPoint.subtract(startPoint);
    double actualHeight = direction.length();
    direction = direction.normalize();

    Vector3 midpoint =
startPoint.toVector().add(endPoint.toVector()).scale(0.5);
}

```

```
// Axis-angle rotation calculation
Vector3 yAxis = new Vector3(0, 1, 0);
Vector3 rotationAxis = yAxis.cross(direction);
double rotationAngle = Math.acos(yAxis.dot(direction));
```

```
// Create rotation matrix (Rodrigues' formula)
double c = Math.cos(rotationAngle);
double s = Math.sin(rotationAngle);
double t = 1 - c;
double x = rotationAxis.x;
double y = rotationAxis.y;
double z = rotationAxis.z;
```

```
Matrix4 rotation = new Matrix4(
    t*x*x + c,  t*x*y - s*z,  t*x*z + s*y,  0,
    t*x*y + s*z,  t*y*y + c,  t*y*z - s*x,  0,
    t*x*z - s*y,  t*y*z + s*x,  t*z*z + c,  0,
    0,          0,          0,          1
);
```

```
Matrix4 translation = Matrix4.translate(midpoint.x, midpoint.y,
midpoint.z);
```

```
Matrix4 scale = Matrix4.scale(1, actualHeight/localHeight, 1);
```

```
this.setTransform(translation.multiply(rotation).multiply(scale));
}
```

```
@Override
public void setMaterial(Material material) {
    this.material = material;
}
```

```
@Override
public Material getMaterial() {
    return this.material;
}
```

```
@Override
```

```
public void setTransform(Matrix4 transform) {
    this.transform = transform;
    this.inverseTransform = transform.inverse();
}

@Override
public Matrix4 getTransform() {
    return this.transform;
}

@Override
public Matrix4 getInverseTransform() {
    return this.inverseTransform;
}

@Override
public Vector3 getNormalAt(Point3 worldPoint) {
    Point3 localHitPoint = inverseTransform.transformPoint(worldPoint);
    Vector3 localNormal;

    double heightFromLocalBase = localHitPoint.y - localBaseCenter.y;

    if (Math.abs(heightFromLocalBase - localHeight) < Ray.EPSILON) {
        localNormal = localAxis;
    } else if (Math.abs(heightFromLocalBase) < Ray.EPSILON) {
        localNormal = localAxis.negate();
    } else {
        Point3 localProjectionOnAxis = new Point3(localBaseCenter.x,
localHitPoint.y, localBaseCenter.z);
        localNormal =
localHitPoint.subtract(localProjectionOnAxis).normalize();
    }

    Matrix4 normalTransformMatrix =
this.inverseTransform.inverseTransposeForNormal();
    return
normalTransformMatrix.transformVector(localNormal).normalize();
}
```

```

@Override
public double intersect(Ray ray) {
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
        inverseTransform.transformVector(ray.getDirection()).normalize();

    double ox = localOrigin.x;
    double oy = localOrigin.y;
    double oz = localOrigin.z;
    double dx = localDirection.x;
    double dy = localDirection.y;
    double dz = localDirection.z;

    // Side surface intersection
    double a = dx * dx + dz * dz;
    double b = 2 * (ox * dx + oz * dz);
    double c = ox * ox + oz * oz - localRadius * localRadius;

    double discriminant = b * b - 4 * a * c;
    double tClosest = -1;

    if (discriminant >= Ray.EPSILON) {
        double sqrtDisc = Math.sqrt(discriminant);
        double t0 = (-b - sqrtDisc) / (2 * a);
        double t1 = (-b + sqrtDisc) / (2 * a);

        if (t0 > t1) {
            double temp = t0;
            t0 = t1;
            t1 = temp;
        }

        if (t0 > Ray.EPSILON) {
            double y0 = oy + t0 * dy;
            if (y0 >= -Ray.EPSILON && y0 <= localHeight + Ray.EPSILON) {
                tClosest = t0;
            }
        }
    }
}

```

```

if (t1 > Ray.EPSILON && (tClosest == -1 || t1 < tClosest)) {
    double y1 = oy + t1 * dy;
    if (y1 >= -Ray.EPSILON && y1 <= localHeight + Ray.EPSILON) {
        tClosest = t1;
    }
}
}

// Cap intersections
if (Math.abs(dy) > Ray.EPSILON) {
    // Bottom cap
    double tBottom = -oy / dy;
    if (tBottom > Ray.EPSILON) {
        double ix = ox + tBottom * dx;
        double iz = oz + tBottom * dz;
        if (ix * ix + iz * iz <= localRadius * localRadius + Ray.EPSILON) {
            if (tClosest == -1 || tBottom < tClosest) {
                tClosest = tBottom;
            }
        }
    }
}

// Top cap
double tTop = (localHeight - oy) / dy;
if (tTop > Ray.EPSILON) {
    double ix = ox + tTop * dx;
    double iz = oz + tTop * dz;
    if (ix * ix + iz * iz <= localRadius * localRadius + Ray.EPSILON) {
        if (tClosest == -1 || tTop < tClosest) {
            tClosest = tTop;
        }
    }
}

return tClosest;
}

/**

```

- * Calculates all intersection intervals between a ray and this finite cylinder.
- * The ray is transformed into the cylinder's local space for calculation.
- * The method checks for intersections with the side, top, and bottom caps.
- * All valid intersections are collected, sorted by t, and paired into intervals.
- * @param ray The ray to test, in world coordinates.
- * @return A list of IntersectionInterval objects. Empty if no intersections.

*/

@Override

```

public List<IntersectionInterval> intersectAll(Ray ray) {
    // 1. Transform the ray into the cylinder's local coordinate system
    Point3 localOrigin = inverseTransform.transformPoint(ray.getOrigin());
    Vector3 localDirection =
    inverseTransform.transformVector(ray.getDirection()).normalize();

    double ox = localOrigin.x;
    double oy = localOrigin.y;
    double oz = localOrigin.z;
    double dx = localDirection.x;
    double dy = localDirection.y;
    double dz = localDirection.z;

    // List to store all intersection t values and their corresponding data
    List<Intersection> hits = new java.util.ArrayList<>();

    // 2. Side surface intersection
    double a = dx * dx + dz * dz;
    double b = 2 * (ox * dx + oz * dz);
    double c = ox * ox + oz * oz - localRadius * localRadius;

    if (a > Ray.EPSILON) { // Avoid division by zero
        double discriminant = b * b - 4 * a * c;
        if (discriminant >= 0) {
            double sqrtDisc = Math.sqrt(discriminant);
            double t0 = (-b - sqrtDisc) / (2 * a);
            double t1 = (-b + sqrtDisc) / (2 * a);
        }
    }
}
```

```

// Check if the intersection points are on the cylinder's height
double y0 = oy + t0 * dy;
double y1 = oy + t1 * dy;

if (t0 > Ray.EPSILON && y0 >= -Ray.EPSILON && y0 <=
localHeight + Ray.EPSILON) {
    Point3 point = ray.pointAtParameter(t0);
    Vector3 normal = getNormalAt(point);
    hits.add(new Intersection(point, normal, t0, this));
}
if (t1 > Ray.EPSILON && y1 >= -Ray.EPSILON && y1 <=
localHeight + Ray.EPSILON) {
    Point3 point = ray.pointAtParameter(t1);
    Vector3 normal = getNormalAt(point);
    hits.add(new Intersection(point, normal, t1, this));
}

// 3. Bottom cap intersection
if (Math.abs(dy) > Ray.EPSILON) {
    double tBottom = -oy / dy;
    if (tBottom > Ray.EPSILON) {
        double ix = ox + tBottom * dx;
        double iz = oz + tBottom * dz;
        if (ix * ix + iz * iz <= localRadius * localRadius + Ray.EPSILON) {
            Point3 point = ray.pointAtParameter(tBottom);
            Vector3 normal = getNormalAt(point);
            hits.add(new Intersection(point, normal, tBottom, this));
        }
    }
}

// 4. Top cap intersection
if (Math.abs(dy) > Ray.EPSILON) {
    double tTop = (localHeight - oy) / dy;
    if (tTop > Ray.EPSILON) {
        double ix = ox + tTop * dx;

```

```

        double iz = oz + tTop * dz;
        if (ix * ix + iz * iz <= localRadius * localRadius + Ray.EPSILON) {
            Point3 point = ray.pointAtParameter(tTop);
            Vector3 normal = getNormalAt(point);
            hits.add(new Intersection(point, normal, tTop, this));
        }
    }
}

// 5. Sort intersections by t value
java.util.Collections.sort(hits, (i1, i2) -> Double.compare(i1.getT(),
i2.getT()));

// 6. Pair intersections into intervals (in-out pairs)
List<IntersectionInterval> intervals = new java.util.ArrayList<>();
for (int i = 0; i < hits.size() - 1; i += 2) {
    Intersection in = hits.get(i);
    Intersection out = hits.get(i + 1);
    intervals.add(new IntersectionInterval(in.getT(), out.getT(), in, out));
}

return intervals;
}

}

// =====
// File: /net/elenamurat/shape/letters/Letter3D.java
// =====

```

```
package net.elena.murat.shape.letters;
```

```
import java.awt.Font;
import java.awt.image.BufferedImage;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;
```

```
import net.elena.murat.math.*;
import net.elena.murat.material.Material;
import net.elena.murat.shape.EMShape;
import net.elena.murat.util.LetterUtils3D;

public class Letter3D implements EMShape {
    private final char letter;
    private final double thickness;
    private Matrix4 transform;
    private Matrix4 inverseTransform;
    private Material material;
    private final LetterUtils3D.LetterMesh mesh;

    public Letter3D(char c) {
        this(c, 16, 1, 1, 0.1, new Font("Arial", Font.BOLD, 32));
    }

    public Letter3D(char letter, int baseSize,
                    double widthScale, double heightScale,
                    double thickness, Font font) {
        this.letter = letter;
        this.thickness = thickness;
        this.transform = Matrix4.identity();
        this.inverseTransform = Matrix4.identity();

        // Original render (slow but correct)
        BufferedImage img = LetterUtils3D.getLetterImage(letter, font,
widthScale, heightScale, baseSize);

        // DEBUG
        try {
            javax.imageio.ImageIO.write(img, "PNG", new
java.io.File("letterImage.png"));
            System.out.println ("Created letterImage.png file.");
        } catch (java.io.IOException ioe)
        {}

        boolean[][] pixelData = LetterUtils3D.getLetterPixelData(img);
        this.mesh = LetterUtils3D.getLetterMeshData(pixelData, thickness);
```

```

}

@Override
public List<IntersectionInterval> intersectAll(Ray worldRay) {
    if (!intersectBoundingBox(worldRay)) {
        return Collections.emptyList();
    }

    Ray localRay = new Ray(
        inverseTransform.transformPoint(worldRay.getOrigin()),
        inverseTransform.transformVector(worldRay.getDirection())
    );

    List<IntersectionInterval> intervals = new ArrayList<>(16);

    for (LetterUtils3D.Face face : mesh.faces) {
        LetterUtils3D.Vertex v1 = mesh.vertices.get(face.v1);
        LetterUtils3D.Vertex v2 = mesh.vertices.get(face.v2);
        LetterUtils3D.Vertex v3 = mesh.vertices.get(face.v3);

        Point3 p1 = new Point3(v1.x, v1.y, v1.z);
        Point3 p2 = new Point3(v2.x, v2.y, v2.z);
        Point3 p3 = new Point3(v3.x, v3.y, v3.z);

        Vector3 edge1 = p2.subtract(p1);
        Vector3 edge2 = p3.subtract(p1);
        Vector3 normal = edge1.cross(edge2);
        double denom = normal.dot(localRay.getDirection());

        if (denom > -Ray.EPSILON && denom < Ray.EPSILON) continue;

        double t =
            normal.dot(p1.toVector().subtract(localRay.getOrigin().toVector())) /
            denom;
        if (t < Ray.EPSILON) continue;

        Point3 hitPoint = localRay.pointAtParameter(t);

        if (isPointInTriangle(hitPoint, p1, p2, p3)) {

```

```

        intervals.add(new IntersectionInterval(t, t,
            new Intersection(hitPoint, normal.normalize(), t, this),
            new Intersection(hitPoint, normal.normalize(), t, this)));
    }
}

intervals.sort((a, b) -> Double.compare(a.tIn, b.tIn));
return intervals;
}

private boolean intersectBoundingBox(Ray worldRay) {
    Point3 boundsMin = new Point3(0, 0, -thickness/2);
    Point3 boundsMax = new Point3(1, 1, thickness/2);
    Point3 localOrigin =
    inverseTransform.transformPoint(worldRay.getOrigin());
    Vector3 localDir =
    inverseTransform.transformVector(worldRay.getDirection());
    double tMin = Double.NEGATIVE_INFINITY;
    double tMax = Double.POSITIVE_INFINITY;

    for (int i = 0; i < 3; i++) {
        double invD = 1.0 / localDir.get(i);
        double t0 = (boundsMin.get(i) - localOrigin.get(i)) * invD;
        double t1 = (boundsMax.get(i) - localOrigin.get(i)) * invD;

        if (invD < 0.0) {
            double temp = t0;
            t0 = t1;
            t1 = temp;
        }

        tMin = Math.max(t0, tMin);
        tMax = Math.min(t1, tMax);

        if (tMax <= tMin) return false;
    }
    return true;
}

```

```

private boolean isPointInTriangle(Point3 p, Point3 a, Point3 b, Point3 c)
{
    Vector3 v0 = c.subtract(a);
    Vector3 v1 = b.subtract(a);
    Vector3 v2 = p.subtract(a);

    double dot00 = v0.dot(v0);
    double dot01 = v0.dot(v1);
    double dot02 = v0.dot(v2);
    double dot11 = v1.dot(v1);
    double dot12 = v1.dot(v2);

    double invDenom = 1.0 / (dot00 * dot11 - dot01 * dot01);
    double u = (dot11 * dot02 - dot01 * dot12) * invDenom;
    double v = (dot00 * dot12 - dot01 * dot02) * invDenom;

    return (u >= 0) && (v >= 0) && (u + v <= 1);
}

```

```

@Override public double intersect(Ray ray) {
    List<IntersectionInterval> intervals = intersectAll(ray);
    return intervals.isEmpty() ? Double.POSITIVE_INFINITY :
intervals.get(0).tIn;
}

```

```

@Override
public Vector3 getNormalAt(Point3 point) {
    Point3 localPoint = inverseTransform.transformPoint(point);
    double minDistance = Double.POSITIVE_INFINITY;
    Vector3 closestNormal = new Vector3(0, 1, 0);

    for (LetterUtils3D.Face face : mesh.faces) {
        LetterUtils3D.Vertex v1 = mesh.vertices.get(face.v1);
        LetterUtils3D.Vertex v2 = mesh.vertices.get(face.v2);
        LetterUtils3D.Vertex v3 = mesh.vertices.get(face.v3);

```

```

        Point3 p1 = new Point3(v1.x, v1.y, v1.z);
        Point3 p2 = new Point3(v2.x, v2.y, v2.z);
        Point3 p3 = new Point3(v3.x, v3.y, v3.z);
    }
}
```

```

Vector3 normal = p2.subtract(p1).cross(p3.subtract(p1)).normalize();
double distance =
Math.abs(normal.dot(localPoint.toVector().subtract(p1.toVector())));

if (distance < minDistance) {
    minDistance = distance;
    closestNormal = normal;
}
}

Vector3 scaledNormal =
inverseTransform.transformNormal(closestNormal);
return new Vector3(
    scaledNormal.x / transform.getScaleX(),
    scaledNormal.y / transform.getScaleY(),
    scaledNormal.z / transform.getScaleZ()
).normalize();
}

@Override public void setMaterial(Material material) { this.material =
material; }
@Override public Material getMaterial() { return material; }

@Override
public void setTransform(Matrix4 transform) {
    this.transform = new Matrix4(transform);
    this.inverseTransform = this.transform.inverse();
}

@Override public Matrix4 getTransform() { return new
Matrix4(transform); }
@Override public Matrix4 getInverseTransform() { return new
Matrix4(inverseTransform); }

}
// =====

```

```
// File: /net/elenamurat/shape/letters/Image3D.java
// =====

package net.elena.murat.shape.letters;

import java.awt.image.BufferedImage;
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

import net.elena.murat.math.*;
import net.elena.murat.material.Material;
import net.elena.murat.shape.EMShape;
import net.elena.murat.util.ImageUtils3D;

public class Image3D implements EMShape {
    private final BufferedImage bimg;
    private final double thickness;
    private Matrix4 transform;
    private Matrix4 inverseTransform;
    private Material material;
    private final ImageUtils3D.ImageMesh mesh;

    public Image3D(BufferedImage c) {
        this(c, 16, 1, 1, 0.1);
    }

    public Image3D(BufferedImage bimg, int baseSize,
                  double widthScale, double heightScale,
                  double thickness) {
        this.bimg = bimg;
        this.thickness = thickness;
        this.transform = Matrix4.identity();
        this.inverseTransform = Matrix4.identity();

        // Original render (slow but correct)
        BufferedImage img = ImageUtils3D.getBufferedImage(bimg,
widthScale, heightScale, baseSize);
```

```

// DEBUG
try {
    javax.imageio.ImageIO.write(img, "PNG", new
java.io.File("x3dImage.png"));
    System.out.println ("Created x3dImage.png file.");
} catch (java.io.IOException ioe)
{}

boolean[][] pixelData = ImageUtils3D.getImagePixelData(img);
this.mesh = ImageUtils3D.getImageMeshData(pixelData, thickness);
}

@Override
public List<IntersectionInterval> intersectAll(Ray worldRay) {
    if (!intersectBoundingBox(worldRay)) {
        return Collections.emptyList();
    }

    Ray localRay = new Ray(
        inverseTransform.transformPoint(worldRay.getOrigin()),
        inverseTransform.transformVector(worldRay.getDirection())
    );

    List<IntersectionInterval> intervals = new ArrayList<>(16);

    for (ImageUtils3D.Face face : mesh.faces) {
        ImageUtils3D.Vertex v1 = mesh.vertices.get(face.v1);
        ImageUtils3D.Vertex v2 = mesh.vertices.get(face.v2);
        ImageUtils3D.Vertex v3 = mesh.vertices.get(face.v3);

        Point3 p1 = new Point3(v1.x, v1.y, v1.z);
        Point3 p2 = new Point3(v2.x, v2.y, v2.z);
        Point3 p3 = new Point3(v3.x, v3.y, v3.z);

        Vector3 edge1 = p2.subtract(p1);
        Vector3 edge2 = p3.subtract(p1);
        Vector3 normal = edge1.cross(edge2);
        double denom = normal.dot(localRay.getDirection());
    }
}

```

```

if (denom > -Ray.EPSILON && denom < Ray.EPSILON) continue;

double t =
normal.dot(p1.toVector().subtract(localRay.getOrigin().toVector())) /
denom;
if (t < Ray.EPSILON) continue;

Point3 hitPoint = localRay.pointAtParameter(t);

if (isPointInTriangle(hitPoint, p1, p2, p3)) {
    intervals.add(new IntersectionInterval(t, t,
        new Intersection(hitPoint, normal.normalize(), t, this),
        new Intersection(hitPoint, normal.normalize(), t, this)));
}
}

intervals.sort((a, b) -> Double.compare(a.tIn, b.tIn));
return intervals;
}

private boolean intersectBoundingBox(Ray worldRay) {
    Point3 boundsMin = new Point3(0, 0, -thickness/2);
    Point3 boundsMax = new Point3(1, 1, thickness/2);
    Point3 localOrigin =
inverseTransform.transformPoint(worldRay.getOrigin());
    Vector3 localDir =
inverseTransform.transformVector(worldRay.getDirection());
    double tMin = Double.NEGATIVE_INFINITY;
    double tMax = Double.POSITIVE_INFINITY;

    for (int i = 0; i < 3; i++) {
        double invD = 1.0 / localDir.get(i);
        double t0 = (boundsMin.get(i) - localOrigin.get(i)) * invD;
        double t1 = (boundsMax.get(i) - localOrigin.get(i)) * invD;

        if (invD < 0.0) {
            double temp = t0;
            t0 = t1;
            t1 = temp;
        }
        if (t0 < tMin) tMin = t0;
        if (t1 > tMax) tMax = t1;
    }
    return tMin < tMax;
}

```

```

    }

    tMin = Math.max(t0, tMin);
    tMax = Math.min(t1, tMax);

    if (tMax <= tMin) return false;
}
return true;
}

private boolean isPointInTriangle(Point3 p, Point3 a, Point3 b, Point3 c)
{
    Vector3 v0 = c.subtract(a);
    Vector3 v1 = b.subtract(a);
    Vector3 v2 = p.subtract(a);

    double dot00 = v0.dot(v0);
    double dot01 = v0.dot(v1);
    double dot02 = v0.dot(v2);
    double dot11 = v1.dot(v1);
    double dot12 = v1.dot(v2);

    double invDenom = 1.0 / (dot00 * dot11 - dot01 * dot01);
    double u = (dot11 * dot02 - dot01 * dot12) * invDenom;
    double v = (dot00 * dot12 - dot01 * dot02) * invDenom;

    return (u >= 0) && (v >= 0) && (u + v <= 1);
}

@Override public double intersect(Ray ray) {
    List<IntersectionInterval> intervals = intersectAll(ray);
    return intervals.isEmpty() ? Double.POSITIVE_INFINITY :
intervals.get(0).tIn;
}

@Override
public Vector3 getNormalAt(Point3 point) {
    Point3 localPoint = inverseTransform.transformPoint(point);
    double minDistance = Double.POSITIVE_INFINITY;

```

```

Vector3 closestNormal = new Vector3(0, 1, 0);

for (ImageUtils3D.Face face : mesh.faces) {
    ImageUtils3D.Vertex v1 = mesh.vertices.get(face.v1);
    ImageUtils3D.Vertex v2 = mesh.vertices.get(face.v2);
    ImageUtils3D.Vertex v3 = mesh.vertices.get(face.v3);

    Point3 p1 = new Point3(v1.x, v1.y, v1.z);
    Point3 p2 = new Point3(v2.x, v2.y, v2.z);
    Point3 p3 = new Point3(v3.x, v3.y, v3.z);

    Vector3 normal = p2.subtract(p1).cross(p3.subtract(p1)).normalize();
    double distance =
        Math.abs(normal.dot(localPoint.toVector().subtract(p1.toVector())));

    if (distance < minDistance) {
        minDistance = distance;
        closestNormal = normal;
    }
}

Vector3 scaledNormal =
inverseTransform.transformNormal(closestNormal);
return new Vector3(
    scaledNormal.x / transform.getScaleX(),
    scaledNormal.y / transform.getScaleY(),
    scaledNormal.z / transform.getScaleZ()
).normalize();
}

@Override public void setMaterial(Material material) { this.material =
material; }
@Override public Material getMaterial() { return material; }

@Override
public void setTransform(Matrix4 transform) {
    this.transform = new Matrix4(transform);
    this.inverseTransform = this.transform.inverse();
}

```

```
    @Override public Matrix4 getTransform() { return new
Matrix4(transform); }
    @Override public Matrix4 getInverseTransform() { return new
Matrix4(inverseTransform); }

}

// =====
// File: /net/elenamurat/lovert/ElenaMuratRayTracer.java
// =====

/*
 * License GNU General Public License v3.0
 * @see <a href="https://www.gnu.org/licenses/gpl-3.0.en.html">GPL v3
License</a>
 */
package net.elena.murat.lovert;

// Java native imports
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.awt.AlphaComposite;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.io.File;
import java.io.IOException;
import java.util.List;
import java.util.Optional;
import java.util.Random;

// Custom imports
import net.elena.murat.shape.*;
import net.elena.murat.shape.letters.*;
import net.elena.murat.material.*;
import net.elena.murat.material.pbr.*;
import net.elena.murat.math.*;
```

```
import net.elena.murat.light.*;
import net.elena.murat.util.*;

/**
 * <h1>TestTracer - Complete Ray Tracing Demo</h1>
 *
 * <div class="block">
 * Full demonstration of the ray tracing engine including all setup steps.
 * </div>
 *
 * <h2>Compilation and Execution</h2>
 * <pre>
 * {@code
 * // Compile with:
 * javac -cp "bin\elenaRT.jar" TestTracer.java
 *
 * // Execute with:
 * java -cp "bin\elenaRT.jar"; TestTracer
 * }
 * </pre>
 *
 * <h2>Complete Implementation</h2>
 * <pre>
 * {@code
package net.elena.murat.lovert;

import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import java.awt.Color;
import java.io.File;
import java.io.IOException;

// Custom imports
import net.elena.murat.shape.*;
import net.elena.murat.lovert.*;
import net.elena.murat.material.*;
import net.elena.murat.material.pbr.*;
import net.elena.murat.math.*;
import net.elena.murat.light.*;
```

```
import net.elena.murat.util.*;  
  
final public class TestTracer {  
  
    private TestTracer() {  
        super();  
    }  
  
    public String toString() {  
        return "TestTracer";  
    }  
  
    final private static void generateSaveRenderedImage(String[] args) throws  
    IOException {  
        // 1. Scene creation  
        Scene scene = new Scene();  
  
        // 2. Ray tracer config (scene, width, height, backgroundColor)  
        ElenaMuratRayTracer rayTracer = new ElenaMuratRayTracer(scene, 800,  
        600, new Color(1f, 1f, 1f));  
  
        // 2-3. Optional  
        rayTracer.setShadowColor (Color.BLUE);  
  
        // 3. Camera setup  
        Camera camera = new Camera();  
        camera.setCameraPosition(new Point3(0, 0, 5));  
        camera.setLookAt(new Point3(0, 0, 0));  
        camera.setUpVector(new Vector3(0, 1, 0));  
        camera.setFov(60.0);  
        camera.setOrthographic(false);  
        camera.setReflective(true);  
        camera.setRefractive(true);  
        camera.setShadowsEnabled(true);  
        camera.setMaxRecursionDepth(2);  
  
        rayTracer.setCamera(camera);  
  
        // 4. Lighting
```

```

scene.addLight(new ElenaMuratAmbientLight(Color.WHITE, 0.5));
scene.addLight(new MuratPointLight(new Point3(-1, 1, 2), Color.WHITE,
1.0));
scene.addLight(new ElenaDirectionalLight(new Vector3(0,-1,0),
Color.WHITE, 1.5));

// 5. Shapes/materials
EMShape ground = new Plane(new Point3(0, 0, 0), new Vector3(0, 1, 0));
ground.setTransform(Matrix4.translate(new Vector3(0, -1.70, 0)));
ground.setMaterial(new CheckerboardMaterial(
new Color(30, 30, 30), new Color(80, 80, 80), 0.4, 1.25, 0.5, 0.3, 8.0,
new Color(255, 230, 180), 0.25, 1.0, 0.0, ground.getInverseTransform()));
scene.addShape(ground);

// 6. Rendering
System.out.println("n==== RENDERING STARTED ====");
long startTime = System.currentTimeMillis();
BufferedImage renderedImage = rayTracer.render();
long endTime = System.currentTimeMillis();
System.out.println("Rendering completed in " + (endTime - startTime) + " ms.");

// 7. Save image
String filename = "images\\example.png";
File outputFile = new File(filename);
ImageIO.write(renderedImage, "png", outputFile);
System.out.println("Image saved: " + outputFile.getAbsolutePath());
}

public static void main(String[] args) {
try {
    generateSaveRenderedImage(args);
} catch (IOException ioe) {
    ioe.printStackTrace();
    System.exit(-1);
}
}
}
}
* }

```

```
* </pre>
*
* @author Murat iNAN, muratsivas76@gmail.com
* @version 1.0
* @since 2023-11-15
*/
```

```
public class ElenaMuratRayTracer {
    private Scene scene;
    private int width;
    private int height;
    private Color backgroundColorAWT;
    private FloatColor backgroundColorFloat;
    private int maxRecursionDepth = 5;

    private Camera camera = new Camera();

    private Point3 cameraPosition;
    private Point3 lookAt;
    private Vector3 upVector;
    private double fov;
    private boolean isOrthographic;
    private boolean isReflective;
    private double orthographicScale = 2.0;

    private Color shadowColor = Color.BLACK;

    public ElenaMuratRayTracer(Scene scene, int width, int height,
        Color backgroundColor) {
        this.scene = scene;
        this.width = width;
        this.height = height;
        this.backgroundColorAWT = backgroundColor;
        this.backgroundColorFloat = new
        FloatColor(backgroundColor.getRed() / 255.0,
            backgroundColor.getGreen() / 255.0,
            backgroundColor.getBlue() / 255.0);
        this.cameraPosition = new Point3(0, 0, 5);
        this.lookAt = new Point3(0, 0, 0);
```

```
this.upVector = new Vector3(0, 1, 0);
this.fov = 60.0;
this.isOrthographic = false;
this.isReflective = true;
}

public void setMaxRecursionDepth(int depth) {
    this.maxRecursionDepth = Math.max(1, Math.min(7, depth));
    camera.setMaxRecursionDepth(this.maxRecursionDepth);
}

public void setCameraPosition(Point3 position) {
    this.cameraPosition = position;
    camera.setCameraPosition(this.cameraPosition);
}

public void setLookAt(Point3 lookAt) {
    this.lookAt = lookAt;
    camera.setLookAt(this.lookAt);
}

public void setUpVector(Vector3 upVector) {
    this.upVector = upVector;
    camera.setUpVector(this.upVector);
}

public void setReflective(boolean isrf) {
    this.isReflective = isrf;
    camera.setReflective(this.isReflective);
}

public void setFov(double fov) {
    this.fov = fov;
    camera.setFov(this.fov);
}

public void setOrthographic(boolean ortho) {
    this.isOrthographic = ortho;
    camera.setOrthographic(this.isOrthographic);
```

```
}

public void setOrthographicScale(double scale) {
    this.orthographicScale = scale;
    camera.setOrthographicScale(this.orthographicScale);
}

public int getMaxRecursionDepth() {
    if(camera == null) {
        return maxRecursionDepth;
    }
    return camera.getMaxRecursionDepth();
}

public void setShadowColor(Color scol) {
    this.shadowColor = scol;
}

public Color getShadowColor() {
    return this.shadowColor;
}

public Camera getCamera() {
    return this.camera;
}

public void setCamera(Camera cmr) {
    if(cmr == null) {
        cmr = new Camera();
    }
    this.camera = cmr;
}

private Ray generateCameraRay(double screenX, double screenY) {
    Vector3 zAxis =
(camera.getCameraPosition()).subtract(camera.getLookAt()).normalize();
    Vector3 xAxis = (camera.getUpVector()).cross(zAxis).normalize();
    Vector3 yAxis = zAxis.cross(xAxis).normalize();
```

```

if (camera.isOrthographic()) {
    double worldScreenWidth = camera.getOrthographicScale();
    double worldScreenHeight = (camera.getOrthographicScale()) /
((double)width/height);
    Point3 rayOrigin = (camera.getCameraPosition())
        .add(xAxis.scale(screenX*worldScreenWidth/2.0))
        .add(yAxis.scale(screenY*worldScreenHeight/2.0));
    return new Ray(rayOrigin, zAxis.negate());
} else {
    double aspectRatio = (double)width/height;
    double tanHalfFov = Math.tan(Math.toRadians(fov)/2);
    Vector3 rayDir = xAxis.scale(screenX*aspectRatio*tanHalfFov)
        .add(yAxis.scale(screenY*tanHalfFov))
        .subtract(zAxis)
        .normalize();
    return new Ray(camera.getCameraPosition(), rayDir);
}
}

```

```

public BufferedImage render() {
    BufferedImage image = new BufferedImage(width, height,
    BufferedImage.TYPE_INT_ARGB);
    Graphics2D g2d = image.createGraphics();

    g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.CLEAR,
    0.0f));
    g2d.fillRect(0, 0, width, height);

    g2d.setComposite(AlphaComposite.SrcOver);
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
    RenderingHints.VALUE_ANTIALIAS_ON);
    g2d.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_INTERPOLATION_BICUBIC);
    g2d.setRenderingHint(RenderingHints.KEY_RENDERING,
    RenderingHints.VALUE_RENDER_QUALITY);

    g2d.setRenderingHint(RenderingHints.KEY_COLOR_RENDERING,
    RenderingHints.VALUE_COLOR_RENDER_QUALITY);

```

```

FloatColor cxx = FloatColor.BLACK;

    final int CERO = 0x0000;

    final double UNO = 1.0;
    final double DOS = 2.0;
    final double HALF = 0.5;

    for (int y = CERO; y < height; y++) {
        for (int x = CERO; x < width; x++) {
            double ndcX = (x + HALF)/width;
            double ndcY = (y + HALF)/height;
            double screenX = DOS*ndcX - UNO;
            double screenY = UNO - DOS*ndcY;

            Ray ray = generateCameraRay(screenX, screenY);

            cxx = traceRay(ray, CERO, UNO);

            image.setRGB(x, y, cxx.toARGB());
        }
    }

    g2d.dispose ();

    return image;
}

//Original traceRay
private FloatColor traceRay(Ray ray, int depth, double attenuationFactor)
{
    // 1. Check depth and attenuation factor
    if ((depth > camera.getMaxRecursionDepth()) || attenuationFactor < 1e-
30) {
        return depth == 0 ? backgroundColorFloat : FloatColor.BLACK;
    }

    // 2. Intersection test
}

```

```

Optional<Intersection> hit = findClosestIntersection(ray);
if (!hit.isPresent()) {
    return depth == 0 ? backgroundColorFloat : FloatColor.BLACK;
}

Intersection intersection = hit.get();
EMShape shape = intersection.getShape();
Material material = shape.getMaterial();
Point3 hitPoint = intersection.getPoint();
Vector3 normal = intersection.getNormal().normalize();

// *** EMOJI BILLBOARD ***
///////////
if (shape instanceof EmojiBillboard) {
    EmojiBillboard billboard = (EmojiBillboard) shape;
    if (!billboard.isVisible()) {
        Color materialColor = material.getColorAt(hitPoint, normal, null,
ray.getOrigin());
        if (materialColor.getAlpha() < 255) {
            Point3 offsetPoint =
hitPoint.add(ray.getDirection().scale(Ray.EPSILON));
            Ray newRay = new Ray(offsetPoint, ray.getDirection());
            return traceRay(newRay, depth, attenuationFactor);
        } else {
            return new FloatColor(
                materialColor.getRed() / 255.0,
                materialColor.getGreen() / 255.0,
                materialColor.getBlue() / 255.0
            );
        }
    }
}
// *** EMOJI BILLBOARD END ***
///////////

if (material == null) {
    if (Math.random() < 0.001) {
        System.out.println("WARNING: NULL MATERIAL converted to

```

```

DIFFUSEMATERIAL: " + shape.toString() + "'");
    }
    material = new DiffuseMaterial(Color.RED);
}

// Normal direction correction
boolean entering = ray.getDirection().dot(normal) < 0;
Vector3 N = entering ? normal : normal.negate();

// 3. Process based on material type
if (material instanceof EmissiveMaterial) {
    // *** EMISSIVE MATERIAL CHECK ***
    EmissiveMaterial emat = (EmissiveMaterial) material;
    return new
FloatColor(emat.getEmissiveColor()).multiply(emat.getEmissiveStrength(
));
} else {
    // *** GENERAL MATERIALS (NON-PBR) ***
    FloatColor finalColor = FloatColor.BLACK;

    // Direct lighting
    Color directLightingColor = calculateDirectLighting(hitPoint, N,
material, ray);
    FloatColor directLightingFloat = new FloatColor(directLightingColor);
    finalColor = finalColor.add(directLightingFloat);

    // Ambient light
    for (Light light : scene.getLights()) {
        if (light instanceof ElenaMuratAmbientLight) {
            Color ambientColor = material.getColorAt(hitPoint, N, light,
ray.getOrigin());
            FloatColor ambientFloat = new FloatColor(ambientColor);
            finalColor = finalColor.add(ambientFloat);
        }
    }

    finalColor = finalColor.multiply(attenuationFactor);

    // Reflection
}

```

```

if (shouldCalculateReflections(material)) {
    Vector3 reflectedDir = ray.getDirection().reflect(N).normalize();
    double newReflectedAttenuation = attenuationFactor *
material.getReflectivity();
    Point3 offsetPoint = hitPoint.add(N.scale(Ray.EPSILON));
    Ray reflectedRay = new Ray(offsetPoint, reflectedDir);
    FloatColor reflectedColor = traceRay(reflectedRay, depth + 1,
newReflectedAttenuation);
    finalColor = finalColor.add(reflectedColor);
}

// Refraction
if (shouldCalculateRefractions(material)) {
    double n1 = entering ? 1.0 : material.getIndexOfRefraction();
    double n2 = entering ? material.getIndexOfRefraction() : 1.0;

    double fresnel = Vector3.calculateFresnel(ray.getDirection(), N, n1,
n2);

    //Optional<Vector3> refractedDir = ray.getDirection().refract(N, n1,
n2); //Original
    Optional<Vector3> refractedDir =
ray.getDirection().refract(entering?normal:normal.negate(), n1, n2);
    //Optional<Vector3> refractedDir =
ray.getDirection().refract(entering?N:N.negate(), n1, n2);

    if (refractedDir.isPresent()) {
        Point3 refractedOffsetPoint =
hitPoint.add(refractedDir.get().scale(Ray.EPSILON));
        //double newRefractedAttenuation = attenuationFactor *
material.getTransparency(); //Original
        double newRefractedAttenuation = attenuationFactor *
material.getTransparency() * (1.0-fresnel);
        Ray refractedRay = new Ray(refractedOffsetPoint,
refractedDir.get());
        FloatColor refractedColor = traceRay(refractedRay, depth + 1,
newRefractedAttenuation);

        if (material instanceof GlassMaterial) {

```

```

Color glassColor =
((GlassMaterial)material).getColorForRefraction();
    FloatColor glassTint = new FloatColor(glassColor);
    refractedColor = refractedColor.multiply(glassTint);
}

if (material instanceof DielectricMaterial) {
    DielectricMaterial dielectric = (DielectricMaterial) material;
    Color filterColor = entering ? dielectric.getFilterColorInside() :
dielectric.getFilterColorOutside();
    FloatColor dielectricTint = new FloatColor(filterColor);
    refractedColor = refractedColor.multiply(dielectricTint);
}

if (material instanceof TextDielectricMaterial) {
    TextDielectricMaterial dielectric = (TextDielectricMaterial)
material;
    Color filterColor = entering ? dielectric.getFilterColorInside() :
dielectric.getFilterColorOutside();
    FloatColor dielectricTint = new FloatColor(filterColor);
    refractedColor = refractedColor.multiply(dielectricTint);
}

if (material instanceof HybridTextMaterial) {
    HybridTextMaterial hybrid = (HybridTextMaterial) material;
    Color filterColor = entering ? hybrid.getFilterColorInside() :
hybrid.getFilterColorOutside();
    FloatColor hybridTint = new FloatColor(filterColor);
    refractedColor = refractedColor.multiply(hybridTint);
}

if (material instanceof DiamondMaterial) {
    Color diamondColor =
((DiamondMaterial)material).getColorForRefraction();
    FloatColor diamondTint = new FloatColor(diamondColor);
    refractedColor = refractedColor.multiply(diamondTint);
}

finalColor = finalColor.add(refractedColor);

```

```

        }
    }

    finalColor = new FloatColor(finalColor.r, finalColor.g, finalColor.b,
finalColor.a);

    return finalColor.clamp01();
}
}

// end of traceRay

private FloatColor calculateLightingForRGB(Point3 point, Vector3
normal, Material material,
Ray ray, double attenuation) {
FloatColor result = new FloatColor(0, 0, 0, 0);

// Direct lighting
try {
    Color direct = calculateDirectLighting(point, normal, material, ray);
    if (direct != null) {
        FloatColor directRGB = new FloatColor(
            direct.getRed() / 255.0,
            direct.getGreen() / 255.0,
            direct.getBlue() / 255.0,
            0.0
        );
        result = result.add(directRGB);
    }
} catch (Exception e) {}

// Ambient lights
for (Light light : scene.getLights()) {
    try {
        if (light instanceof ElenaMuratAmbientLight) {
            Color ambient = material.getColorAt(point, normal, light,
ray.getOrigin());
            if (ambient != null) {
                FloatColor ambientRGB = new FloatColor(
                    ambient.getRed() / 255.0,

```

```

        ambient.getGreen() / 255.0,
        ambient.getBlue() / 255.0,
        0.0
    );
    result = result.add(ambientRGB);
}
}
} catch (Exception e) {}
}

return result.multiply(attenuation);
}

/***
 * Applies colored absorption to a light color based on tint.
 * Preserves the alpha (transparency) of the original color.
 *
 * @param originalColor The incoming light color (with alpha)
 * @param tintColor The glass/material tint color (used for absorption)
 * @return Absorbed color with original alpha preserved
 */
public FloatColor applyAbsorption(FloatColor originalColor, Color
tintColor) {
    // 1. Convert tint color to FloatColor
    FloatColor tint = new FloatColor(tintColor);

    // 2. Absorption coefficients (higher tint → less absorption)
    // - Red tint (high R) → absorbs less red
    double absorptionR = 1.0 - tint.r;
    double absorptionG = 1.0 - tint.g;
    double absorptionB = 1.0 - tint.b;
    //double absorptionA = 1.0 - tint.a;//originalColor.a;

    // 3. Base transmission (e.g., 97% light passes through)
    final double TRANSMISSION = 0.97; // 3% base absorption

    // 4. Apply absorption to RGB components
    double r = originalColor.r * (1.0 - absorptionR * (1.0 -
TRANSMISSION));

```

```

    double g = originalColor.g * (1.0 - absorptionG * (1.0 -
TRANSMISSION));
    double b = originalColor.b * (1.0 - absorptionB * (1.0 -
TRANSMISSION));
    //double a = originalColor.a * (1.0 - absorptionA * (1.0 -
TRANSMISSION));

    // 5. Preserve original alpha
    //double a = originalColor.a;

    return new FloatColor(r, g, b).clamp01();
}

// New helper methods (CAMERA CONTROLLED)
private boolean shouldCalculateReflections(Material material) {
    return camera.isReflective()
    && material.getReflectivity() > Ray.EPSILON;
}

private boolean shouldCalculateRefractions(Material material) {
    return camera.isRefractive()
    && material.getTransparency() > Ray.EPSILON;
}

private boolean shouldCalculateShadows() {
    return camera.isShadowsEnabled();
}

private Color calculateDirectLighting(Point3 point, Vector3 normal,
Material material, Ray ray) {
    Color directLightingColor = new Color(0, 0, 0);

    for (Light light : scene.getLights()) {
        if (light instanceof ElenaMuratAmbientLight) continue;

        Vector3 lightDir = null;
        double distance = Double.POSITIVE_INFINITY;

        if (light instanceof MuratPointLight) {

```

```

MuratPointLight ptLight = (MuratPointLight) light;
lightDir = ptLight.getPosition().subtract(point).normalize();
distance = ptLight.getPosition().subtract(point).length();
} else if (light instanceof PulsatingPointLight) {
PulsatingPointLight pulsatingLight = (PulsatingPointLight) light;
lightDir = pulsatingLight.getPosition().subtract(point).normalize();
distance = pulsatingLight.getDistanceTo(point);
} else if (light instanceof ElenaDirectionalLight) {
ElenaDirectionalLight dl = (ElenaDirectionalLight) light;
lightDir = dl.getDirection().negate().normalize();
distance = Double.POSITIVE_INFINITY;
} else if (light instanceof SpotLight) {
SpotLight s = (SpotLight) light;
lightDir = s.getPosition().subtract(point).normalize();
distance = s.getPosition().subtract(point).length();
} else if (light instanceof FractalLight) {
FractalLight f = (FractalLight) light;
lightDir = f.getPosition().subtract(point).normalize();
distance = f.getPosition().subtract(point).length();
} else if (light instanceof BlackHoleLight) {
BlackHoleLight bh = (BlackHoleLight) light;
lightDir = bh.getPosition().subtract(point).normalize();
distance = bh.getPosition().subtract(point).length();
} else if (light instanceof BioluminescentLight) {
BioluminescentLight bio = (BioluminescentLight) light;
lightDir = bio.getDirectionAt(point);
distance = bio.getClosestDistance(point);
}

if (lightDir == null) continue;

if (material instanceof TransparentPNGMaterial ||
    material instanceof TransparentColorMaterial ||
    material instanceof GhostTextMaterial) {
    Color contribution = material.getColorAt(point, normal, light,
ray.getOrigin());
    directLightingColor = ColorUtil.addSafe(directLightingColor,
contribution);
    continue;
}

```

```

    }

    // Only calculate direct lighting if not in shadow
    if (!shouldCalculateShadows() ||
        !isInShadow(point.add(normal.scale(Ray.EPSILON)), lightDir,
distance)) {
        Color contribution = material.getColorAt(point, normal, light,
ray.getOrigin());
        directLightingColor = ColorUtil.addSafe(directLightingColor,
contribution);
    } else {
        // Shadow color apply
        directLightingColor =
ColorUtil.applyShadowColor(directLightingColor, shadowColor);
    }
}

return directLightingColor;
}

private Optional<Intersection> findClosestIntersection(Ray ray) {
    EMShape closestShape = null;
    double closestDist = Double.POSITIVE_INFINITY;

    for (EMShape shape : scene.getShapes()) {
        double dist = shape.intersect(ray);
        if (dist > Ray.EPSILON && dist < closestDist) {
            closestDist = dist;
            closestShape = shape;
        }
    }

    if (closestShape == null) return Optional.empty();

    Point3 hitPoint = ray.pointAtParameter(closestDist);
    Vector3 normal = closestShape.getNormalAt(hitPoint);
    return Optional.of(new Intersection(hitPoint, normal, closestDist,
closestShape));
}

```

```
private boolean isInShadow(Point3 point, Vector3 lightDir, double
lightDistance) {
    Ray shadowRay = new Ray(point, lightDir);
    Optional<Intersection> shadowHit =
findClosestIntersection(shadowRay);
    return shadowHit.isPresent() && shadowHit.get().getDistance() <
lightDistance - Ray.EPSILON;
}

public static void main(String[] args) {
    // Classes for compiling all packages easily
    UnionCSG ucsg = null;
    IntersectionCSG icsg = null;
    DifferenceCSG dcsg = null;
    PulsatingPointLight ppl = null;
    BioluminescentLight blm = null;
    BlackHoleLight bhl = null;
    FractalLight flt = null;
    SpotLight spli = null;
    CircleTextureMaterial ctm = null;
    SquaredMaterial sm = null;
    SolidColorMaterial scm = null;
    StripedMaterial stma = null;
    RectangularPrism rp = null;
    LambertMaterial lm = null;
    TriangleMaterial trm = null;
    CheckerboardMaterial cm = null;
    TexturedCheckerboardMaterial tcimo = null;
    GhostTextMaterial ghost = null;
    DiamondMaterial diamond = null;
    TexturedPhongMaterial tpma = null;
    ThresholdMaterial thresold = null;
    BrightnessMaterial brighto = null;
    ContrastMaterial contrast = null;
    SphereWordTextureMaterial spwordmat = null;
    MosaicMaterial mosmat = null;
    CrystalMaterial crysmat = null;
    PolkaDotMaterial polkamat = null;
```

```
OrbitalMaterial orbit = null;
MultiMixMaterial multimix = null;
SuperBrightDebugMaterial sbdm = null;
ImageTextureMaterial itm = null;
MetallicMaterial mmt = null;
RoughMaterial romu = null;
RubyMaterial rubimo = null;
DielectricMaterial dielectric = null;
ObsidianMaterial obsidian = null;
EmeraldMaterial emerald = null;
EdgeLightColorMaterial edgelit = null;
InvertLightColorMaterial ilight = null;
PixelArtMaterial pam = null;
HolographicDiffractionMaterial hdm = null;
HolographicPBRMaterial hcpm = null;
ChromePBRMaterial chrpmt = null;
WaterPBRMaterial wpmcv = null;
BlackHoleMaterial bhm = null;
FractalBarkMaterial fbm = null;
StarfieldMaterial sfm = null;
ProceduralFlowerMaterial pfmtr = null;
DamaskCeramicMaterial dcm = null;
LavaFlowMaterial lfm = null;
WaterRippleMaterial wrm = null;
QuantumFieldMaterial qfm = null;
StainedGlassMaterial sgm = null;
RandomMaterial rmat = null;
PhongMaterial phm = null;
PhongTextMaterial ptomme = null;
HybridTextMaterial hibrid = null;
    TransparentColorMaterial tecome = null;
PhongElenaMaterial pem = null;
GlassMaterial glsmat = null;
GoldPBRMaterial gpbrmat = null;
CeramicTilePBRMaterial ctpm = null;
ClassicTilePBRMaterial glptm = null;
SilverPBRMaterial silvom = null;
MarblePBRMaterial mbppp = null;
SolidCheckerboardMaterial nrcbm = null;
```

```
MarbleMaterial mbtt = null;
DewDropMaterial ddm = null;
HexagonalHoneycombMaterial hhcm = null;
OpticalIllusionMaterial oim = null;
Cube cube = null;
Cone cone = null;
TransparentPlane ptl = null;
Ellipsoid els = null;
Triangle tri = null;
Plane plane = null;
Cylinder cll = null;
Rectangle3D r3d = null;
Triangle tris = null;
Torus torus = null;
Box box = null;
TorusKnot toruskn = null;
Crescent ccr = null;
MaterialUtils mut = null;
ResizeImage rszz = null;
Hyperboloid hypb = null;
PBRCapableMaterial pbrcm = null;
PlasticPBRMaterial plasmat = null;
CopperPBRMaterial coppmat = null;
WoodPBRMaterial wpmt = null;
DiagonalCheckerMaterial dicem = null;
RectangleCheckerMaterial rcm = null;
DiffuseMaterial dmm = null;
CrystalClearMaterial ccm = null;
PlatinumMaterial plm = null;
WoodMaterial wmt = null;
ElenaTextureMaterial etm = null;
GradientTextMaterial grtm = null;
GradientImageTextMaterial gritm = null;
TurkishTileMaterial trtilem = null;
NorwegianRoseMaterial nwmat = null;
NordicWoodMaterial nomat = null;
CoffeeFjordMaterial cofij = null;
NorthernLightMaterial nilon = null;
CarpetTextureMaterial cetome = null;
```

AnodizedMetalMaterial anotem = null;
AnodizedTextMaterial anodizedWord = null;
AmberMaterial amber = null;
XRayMaterial xray = null;
ProceduralCloudMaterial procemo = null;
LinearGradientMaterial ligram = null;
RadialGradientMaterial radimat = null;
VikingMetalMaterial vkgmt = null;
TextDielectricMaterial tdiele = null;
TransparentEmojiMaterial trnsEmoj = null;
TransparentPNGMaterial tpng = null;
NonScaledTransparentPNGMaterial nonsctrns = null;
TransparentEmissivePNGMaterial tepng = null;
WordMaterial womat = null;
EmojiBillboard ebilbo = null;
HotCopperMaterial hocop = null;
NordicWeaveMaterial nqwmt = null;
RuneStoneMaterial rosom = null;
AuroraCeramicMaterial acome = null;
FjordCrystalMaterial fjome = null;
RosemalmingMaterial risomel = null;
TelemarkPatternMaterial telemol = null;
BrunostCheeseMaterial buconi = null;
VikingRuneMaterial viruni = null;
KilimRosemalmingMaterial kirose = null;
CalligraphyRuneMaterial callimo = null;
TulipFjordMaterial tjfo = null;
HamamSaunaMaterial hasuna = null;
SultanKingMaterial sukemat = null;
GradientChessMaterial gcm = null;
SmartGlassMaterial sglas = null;
HologramDataMaterial holdam = null;
WaterfallMaterial wemf = null;
PureWaterMaterial pwm = null;
ReflectiveMaterial refoma = null;
LightningMaterial ligoma = null;
FractalFireMaterial ffm = null;
NeutralMaterial nnm = null;
TextureMaterial timam = null;

```

ColorUtil cutil = null;
Letter3D l3d = null;
Image3D i3d = null;

// 1. Create Scene
Scene scene = new Scene();

// 2. Create Ray Tracer
int imageWidth = 800;
int imageHeight = 600;
Color rendererBackgroundColor = new Color(0.2f, 0.2f, 0.2f);

// Create ElenaMuratRayTracer
ElenaMuratRayTracer rayTracer = new ElenaMuratRayTracer(scene,
imageWidth, imageHeight, rendererBackgroundColor);

// 3. Adjust ray tracer values
Camera cmra = new Camera();

cmra.setCameraPosition(new Point3(0, 1, 8));
cmra.setLookAt(new Point3(0, 0, -3));
cmra.setUpVector(new Vector3(0, 1, 0));
cmra.setFov(60.0);
cmra.setMaxRecursionDepth(2); // Max recursion depth
cmra.setOrthographic(false);
cmra.setReflective(true); // Reflections enabled
cmra.setRefractive(true); // Refractions enabled
cmra.setShadowsEnabled(true); // Shadows enabled

rayTracer.setCamera(cmra);

// 4. Create and add lights
// Ambient (More bluish and stronger)
scene.addLight(new ElenaMuratAmbientLight(new Color(220, 225,
255), 2.5));

// Main light (Softer but stronger)
scene.addLight(new ElenaDirectionalLight(
new Vector3(-0.7, -1, -0.4).normalize(),

```

```

    new Color(255, 245, 235), // More neutral white
    3.8
));
// Fill light (Wider area)
scene.addLight(new MuratPointLight(
    new Point3(2, 3, 1),
    new Color(230, 235, 255),
    3.5
));
// Back light (More pronounced)
scene.addLight(new ElenaDirectionalLight(
    new Vector3(0.3, 0.3, 1).normalize(), // Direction adjustment
    new Color(255, 255, 255),
    2.0
));
// Specular highlight light
scene.addLight(new MuratPointLight(
    new Point3(-1, 2, 0.5),
    new Color(255, 255, 240),
    2.3
));
// Global illumination (Subtle touch for entire scene)
scene.addLight(new ElenaDirectionalLight(
    new Vector3(0.2, -0.3, 0.1).normalize(),
    new Color(210, 220, 255),
    1.5
));
// 5. Create shapes with materials and add to scene
// --- Four Basic Material Spheres ---
// a. Gold Sphere
Sphere goldSphere = new Sphere(0.7); // Create with radius only
goldSphere.setMaterial(new GoldMaterial());
goldSphere.setTransform(Matrix4.translate(new Vector3(-1.5, 0.5,
0))); // Set position

```

```

scene.addShape(goldSphere);

// b. Silver Sphere
Sphere silverSphere = new Sphere(0.7);
silverSphere.setMaterial(new SilverMaterial());
silverSphere.setTransform(Matrix4.translate(new Vector3(1.5, 0.5, 0)));
scene.addShape(silverSphere);

// c. Copper Sphere
CopperMaterial copperMat = new CopperMaterial();
Sphere copperSphere = new Sphere(0.7);
copperSphere.setMaterial(copperMat);
copperSphere.setTransform(Matrix4.translate(new Vector3(-0.75, -1.0, -1.0)));
scene.addShape(copperSphere);

// d. Emissive Sphere
Sphere emissiveSphere = new Sphere(0.7);
emissiveSphere.setMaterial(new EmissiveMaterial(new Color(255, 100, 0), 3.0));
emissiveSphere.setTransform(Matrix4.translate(new Vector3(0.75, -1.0, -1.0)));
scene.addShape(emissiveSphere);

// --- Bump Mapped Sphere ---
BufferedImage bumpImage = null;
ImageTexture bumpTexture = null;
try {
    bumpImage = ImageIO.read(new File("textures\\elena.png"));
    bumpTexture = new ImageTexture(bumpImage, 1.0);
    System.out.println("Normal map loaded successfully.");
} catch (IOException e) {
    System.err.println("ERROR: Normal map could not be loaded: " +
e.getMessage());
    e.printStackTrace();
}

Material bumpyMaterial = null;
Sphere bumpySphere = new Sphere(0.8);

```

```

bumpySphere.setTransform(Matrix4.translate(new Vector3(0, 0, -2)));

if (bumpTexture != null) {
    bumpyMaterial = new BumpMaterial(
        new LambertMaterial(new Color(100, 150, 200)), // Base material
        (blue Lambertian)
        bumpTexture,
        1.0, // Bump strength
        5.0, // UV scale
        bumpySphere.getInverseTransform()
    );
} else {
    bumpyMaterial = new LambertMaterial(new Color(100, 150, 200));
}
bumpySphere.setMaterial(bumpyMaterial);
scene.addShape(bumpySphere);

// --- Floor ---
Plane floorPlane = new Plane(new Point3(0, 0, 0), new Vector3(0, 1,
0));
floorPlane.setTransform(Matrix4.translate(new Vector3(0, -1.7, 0)));
CheckerboardMaterial floorMaterial = new CheckerboardMaterial(
    new Color(100, 100, 100), // Dark gray
    new Color(200, 200, 200), // Light gray
    4.0, // Scale: 4 squares per unit length
    0.1, // ambientCoefficient
    0.8, // diffuseCoefficient
    0.2, // specularCoefficient
    10.0, // shininess
    Color.WHITE, // specularColor
    0.0, // reflectivity
    1.0, // indexOfRefraction
    0.0, // transparency
    floorPlane.getInverseTransform()
);
floorPlane.setMaterial(floorMaterial);
scene.addShape(floorPlane);

// Right Wall Plane (X=5, normal (-1,0,0))

```

```

Plane rightWallPlane = new Plane(new Point3(0, 0, 0), new Vector3(-1,
0, 0));
Matrix4 rightWallTransform = Matrix4.translate(new Vector3(5, 0, 0));
rightWallPlane.setTransform(rightWallTransform);
SquaredMaterial rightWallMaterial = new SquaredMaterial(
    new Color(0.3f, 0.0f, 0.0f), // Dark red
    new Color(1.0f, 0.0f, 0.0f), // Bright red
    4.0, // Square size
    0.1, 0.7, 0.8, 50.0, Color.WHITE, // Phong parameters
    0.0, 1.0, 0.0,
    rightWallPlane.getInverseTransform()
);
rightWallPlane.setMaterial(rightWallMaterial);
scene.addShape(rightWallPlane);

// Left Wall Plane (X=-5, normal (1,0,0))
Plane leftWallPlane = new Plane(new Point3(0, 0, 0), new Vector3(1, 0,
0));
Matrix4 leftWallTransform = Matrix4.translate(new Vector3(-5, 0, 0));
leftWallPlane.setTransform(leftWallTransform);
SquaredMaterial leftWallMaterial = new SquaredMaterial(
    new Color(0.0f, 0.3f, 0.0f), // Dark green
    new Color(0.0f, 1.0f, 0.0f), // Bright green
    4.0, // Square size
    0.1, 0.7, 0.8, 50.0, Color.WHITE, // Phong parameters
    0.0, 1.0, 0.0,
    leftWallPlane.getInverseTransform()
);
leftWallPlane.setMaterial(leftWallMaterial);
scene.addShape(leftWallPlane);

// 6. Render image
System.out.println("Render process starting...");
long startTime = System.currentTimeMillis();

BufferedImage renderedImage = rayTracer.render();

long endTime = System.currentTimeMillis();
System.out.println("Render process completed. Time: " + (endTime -

```

```
startTime) + " ms");

// 7. Save image
try {
    File outputFile = new File("images\\rendered_scene.png");
    ImageIO.write(renderedImage, "png", outputFile);
    System.out.println("Image successfully saved: " +
outputFile.getAbsolutePath());
} catch (IOException e) {
    System.err.println("An error occurred while saving the image: " +
e.getMessage());
    e.printStackTrace();
}
}
```

```
// =====
// File: /net/elenamurat/lovert/Camera.java
// =====
```

```
package net.elena.murat.lovert;

import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

public class Camera
extends Object
implements java.io.Serializable {

    private Point3 cameraPosition=new Point3 (0, 1, 5);
    private Point3 lookAt=new Point3 (0, 0, -1);

    private Vector3 upVector=new Vector3 (0, 1, 0);

    private boolean ortographic=false;

    private double fov=0.0;
```

```
private double orthographicScale=1.0;  
  
private int maxRecursionDepth=3;  
  
private boolean isReflective=true;  
private boolean isRefractive=true;  
private boolean shadowsEnabled = true;  
  
public Camera () {  
    super ();  
}  
  
public String toString () {  
    return "Camera Settings...";  
}  
  
// Getters  
public Point3 getCameraPosition () {  
    return cameraPosition;  
}  
  
public Point3 getLookAt () {  
    return lookAt;  
}  
  
public Vector3 getUpVector () {  
    return upVector;  
}  
  
public double getFov () {  
    return fov;  
}  
  
public int getMaxRecursionDepth () {  
    return maxRecursionDepth;  
}  
  
public double getOrthographicScale () {
```

```
    return this.ortographicScale;
}

public boolean isOrthographic () {
    return this.ortographic;
}

public boolean isReflective () {
    return this.isReflective;
}

public boolean isRefractive () {
    return this.isRefractive;
}

public boolean isShadowsEnabled() {
    return shadowsEnabled;
}

// Setters
public void setCameraPosition (Point3 pv) {
    this.cameraPosition=pv;
}

public void setLookAt (Point3 pv) {
    this.lookAt=pv;
}

public void setUpVector (Vector3 vv) {
    this.upVector=vv;
}

public void setFov (double dbl) {
    this.fov=dbl;
}

public void setMaxRecursionDepth (int mr) {
    this.maxRecursionDepth=mr;
}
```

```
public void setOrthographicScale (double scl) {  
    this.ortographicScale=scl;  
}  
  
public void setOrthographic (boolean bb) {  
    this.ortographic=bb;  
}  
  
public void setReflective (boolean rb) {  
    this.isReflective=rb;  
}  
  
public void setRefractive (boolean rb) {  
    this.isRefractive=rb;  
}  
  
public void setShadowsEnabled(boolean enabled) {  
    this.shadowsEnabled = enabled;  
}  
}
```

```
// ======  
// File: /net/elenamurat/lovert/Scene.java  
// ======
```

```
package net.elena.murat.lovert;  
  
import java.util.ArrayList;  
import java.util.List;  
import java.util.Optional;  
  
//custom imports  
import net.elena.murat.math.Intersection;  
import net.elena.murat.math.Ray;  
import net.elena.murat.shape.EMShape;  
import net.elena.murat.light.Light;
```

```
import net.elena.murat.math.Point3;
import net.elena.murat.math.Vector3;

/**
 * Represents a 3D scene containing shapes and lights for ray tracing.
 * Handles intersection tests and light management.
 */
public class Scene {
    private final List<EMShape> shapes = new ArrayList<>();
    private final List<Light> lights = new ArrayList<>();

    /**
     * Adds a shape to the scene
     * @param shape The shape to add
     */
    public void addShape(EMShape shape) {
        shapes.add(shape);
    }

    /**
     * Adds a light source to the scene
     * @param light The light to add
     */
    public void addLight(Light light) {
        lights.add(light);
    }

    public List<EMShape> getShapes() {
        return new ArrayList<>(shapes); // Return copy for immutability
    }

    public List<Light> getLights() {
        return new ArrayList<>(lights); // Return copy for immutability
    }

    /**
     * Clears all shapes from the scene.
     */
    public void clearShapes() {
```

```

        shapes.clear();
    }

/***
 * Clears all lights from the scene.
 */
public void clearLights() {
    lights.clear();
}

/***
 * Finds the closest ray-object intersection in the scene
 * @param ray The ray to test
 * @return Optional containing closest intersection if found
 */
public Optional<Intersection> intersect(Ray ray) {
    return intersect(ray, null); // No shape excluded by default
}

/***
 * Finds the closest ray-object intersection excluding a specific shape
 * @param ray The ray to test
 * @param excludeShape Shape to exclude from intersection tests
 * @return Optional containing closest intersection if found
 */
public Optional<Intersection> intersect(Ray ray, EMShape
excludeShape) {
    EMShape closestShape = null;
    double minDistance = Double.POSITIVE_INFINITY;
    Point3 closestHitPoint = null;

    for (EMShape shape : shapes) {
        if (shape == excludeShape) {
            continue;
        }

        double distance = shape.intersect(ray);

        if (distance < minDistance && distance > Ray.EPSILON) {

```

```

        minDistance = distance;
        closestShape = shape;
        closestHitPoint = ray.pointAtParameter(distance);
    }
}

if (closestShape != null) {
    Vector3 normal = closestShape.getNormalAt(closestHitPoint);
    // Normal orientation will be fixed during shading
    return Optional.of(new Intersection(
        closestHitPoint,
        normal,
        minDistance,
        closestShape
    ));
}
return Optional.empty();
}

public boolean intersects(Ray ray, double maxDistance) {
    for (EMShape shape : shapes) {
        double distance = shape.intersect(ray);
        if (distance > Ray.EPSILON && distance < maxDistance) {
            return true;
        }
    }
    return false;
}

/**
 * Checks if a point is visible from any light source (for soft shadows)
 * @param point The point to test
 * @return Number of lights visible from the point
 */
public int getVisibleLightCount(Point3 point) {
    int count = 0;
    for (Light light : lights) {
        if (light.isVisibleFrom(point, this)) {
            count++;
        }
    }
}

```

```
    }
}
return count;
}
}
// END of PROJECT
```